

Tutorial Index

The tutorials on this page may contain mistakes, poor commenting, and should not be considered the best resource to learn OpenGL from. What you do with the code is up to you. I am merely trying to make the learning process a little easier for those people new to OpenGL. If you are serious about learning OpenGL, you should spend the money and invest in the OpenGL Red Book (ISBN 0-201-46138-2) and OpenGL Blue Book (ISBN 0-201-46140-4). I have the second edition of each book, and although they can be difficult for the new OpenGL programmer to understand, they are by far the best books written on the subject of OpenGL. Another book I would recommend is the OpenGL Superbible, although opinions vary. It is also important that you have a solid understanding of the language you plan to use. Although I do comment the non-GL lines, I am self-taught, and may not always write proper or even good code. It's up to you to take what you have learned from this site and apply it to projects of your own. Play around with the code, read books, ask me questions if need be. Once you have surpassed the code on this site or even before, check out more professional sites, such as Nate's Programming Page or OpenGL.org. Although Nate's site is inactive at the moment, it contains tons of excellent example programs, that are well written, and really show off what OpenGL is capable of. Also be sure to visit the many OpenGL links on my page. Each site I link to is an incredible asset the OpenGL community. Most of these sites are run by talented individuals that not only know their GL, they also program alot better than I do. Please keep all of this in mind while browsing my site. I hope you enjoy what I have to offer, and hope to see projects created by yourself in the near future!

One final note, if you see code that you feel is to similar to someone else's code, please contact me. I assure you, any code I borrow from or learn from either comes from the MSDN or from sites created to help teach people in a similar way that my site teaches GL. I never intentionally take code, and never would without giving the proper person credit. There may be instances where I get code from a free site not knowing that site took it from someone else, so if that happens, please contact me. I will either rewrite the code, or remove it from my program. Most the code should be original however, I only borrow when I absolutely have no idea how to accomplish something, and even then I make sure I understand the code before I decide to include it in my program. If you spot mistakes in any of the tutorials, no matter how tiny the mistake may be, please let me know.

One important thing to note about my base code is that it was written in 1997. It has undergone many changes, and it is definitely not borrowed from any other sites. It will more than likely be altered in the future. If I am not the one that modifies it, the person responsible for the changes will be credited.



Setting Up OpenGL In MacOS:

This is not a tutorial, but a step by step walkthrough done by Tony Parker on how to install OpenGL and Glut under Mac OS. Tony has kindly ported the OpenGL tutorials I've done to Mac OS with GLUT. I hope everyone enjoys the ports.

I know alot of people have asked for Mac ports so support Tony by telling him how much you enjoy the ports. Without his work converting the projects there wouldn't be a Mac port.

Setting Up OpenGL In Solaris:

This is not a tutorial, but a step by step walkthrough



done by Lakmal Gunasekara on how to install OpenGL and Glut under Solaris. Lakmal has kindly ported most of the OpenGL tutorials I've done to both Irix and Solaris. I hope everyone enjoys the ports.

If you'd like to port the code to another OS or Language, please contact me, and let me know. Before you start porting, keep in mind that I'd prefer all the code to be ported, rather than just a few of the tutorials. That way, people learning from a port can learn at the same rate as the VC guys.



This tutorial was written by Raal Goff and will teach you how to get OpenGL working on MacOS X using GLUT. Nothing really different aside from the headers and environment, but definitely a useful resource for anyone using this cool looking new OS. If you enjoy the information email Raal and let him know.

Now that OpenGL is the API of choice for Mac users, I hope to see alot more demos, projects and games from all of you Mac users! It's good to see Apple supporting such a strong API. This tutorial may also be useful to those of you interested in using GLUT instead of the framework from lesson 1.

Unit Computer Image: Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Control Flow Soliest as down to we be description. Soliest as down to we be description. Soliest as down to we be description. Soliest as down to we be description.

Setting Up An OpenGL Window:

In this tutorial, I will teach you how to set up, and use OpenGL in a Windows environment. The program you create in this tutorial will display an empty OpenGL window, switch the computer into fullscreen or windowed mode, and wait for you to press ESC or close the Window to exit. It doesn't sound like much, but this program will be the framework for every other tutorial I release in the next while.

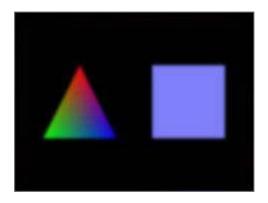
It's very important to understand how OpenGL works, what goes into creating an OpenGL Window, and how to write simple easy to understand code. You can download the code at the end of the tutorial, but I definitely recommend you read over the tutorial at least once, before you start programming in OpenGL.

Your First Polygon:

Using the source code from the first tutorial, we will now add code to create a Triangle, and a Square on the screen. I know you're probably thinking to yourself "a triangle and square... oh joy", but it really is a BIG deal. Just about everything you create in OpenGL will be created out of triangles and squares. If you don't understand how to create a simple little triangle in Three Dimensional space, you'll be completely lost down the road. So read through this chapter and learn.

Once you've read through this chapter, you should understand the X axis, Y axis and Z axis. You will learn about translation left, right, up, down, into and out of the screen. You should understand how to place an object on the screen exactly where you want it to be. You will



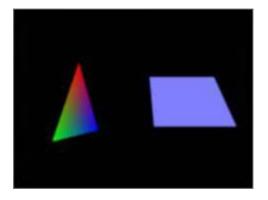


also learn a bit about the depth buffer (placing objects into the screen).

Colors:

Expanding on the second tutorial I will teach you how to create spectacular colors in OpenGL with very little effort. You will learn about both flat coloring and smooth coloring. The triangle on the left uses smooth coloring. The square on the right is using flat coloring. Notice how the colors on the triangle blend together.

Color adds alot to an OpenGL project. By understanding both flat and smooth coloring, you can greatly enhance the way your OpenGL demos look.



Rotation:

Moving right along. In this tutorial I'll teach you how to rotate both the triangle and the quad. The triangle will rotate on the Y axis, and the quad will rotate on the X axis. This tutorial will introduce 2 variables. rtri is used to store the angle of the triangle, and rquad will store the angle of the quad.

It's easy to create a scene made up of polygons. Adding motion to those object makes the scene come alive. In later tutorials I'll teach you how to rotate an object around a point on the screen causing the object to move around the screen rather than spin on its axis.

Solid Objects:

Now that we have setup, polygons, quads, colors and rotation figured out, it's time to build 3D objects. We'll build the objects using polygons and quads. This time we'll expand on the last tutorial, and turn the triangle into a colorful pyramid, and turn the square into a solid cube. The pyramid will use blended colors, the cube will have a different color for each face.

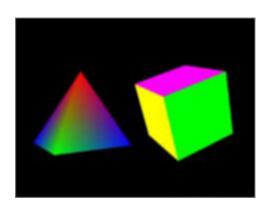
Building an object in 3D can be very time consuming, but the results are usually worth it. Your imagination is the limit!



Texture Mapping:

You asked for it, so here it is... Texture Mapping!!! In this tutorial I'll teach you how map a bitmap image onto the six side of a cube. We'll use the GL code from lesson one to create this project. It's easier to start with an empty GL window than to modify the last tutorial.

You'll find the code from lesson one is extremely valuable when it comes to developing a project quickly. The code in lesson one sets everything up for you, all you have to do is concentrate on programming the effect (s).



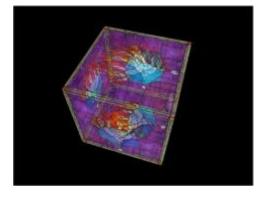


Texture Filters, Lighting & Keyboard Control:

Ok, I hope you've been understanding everything up till now, because this is a huge tutorial. I'm going to attempt to teach you 2 new ways to filter your textures, simple lighting, keyboard control, and probably more :) If you don't feel confident with what you've learned up to this lesson, go back and review. Play around with the code in the other tutorials. Don't rush. It's better to take your time and learn each lesson well, than to jump in, and only know enough to get things done.

* Blending:

There was a reason for the wait. A fellow programmer from the totally cool site Hypercosm, had asked if he could write a tutorial on blending. Lesson eight was going to be a blending tutorial anyways. So the timing was perfect! This tutorial expands on lesson seven. Blending is a very cool effect... I hope you all enjoy the tutorial. The author of this tutorial is Tom Stanis. He's put alot of effort into the tutorial, so let him know what you think. Blending is not an easy topic to cover.





Moving Bitmaps In 3D Space:

This tutorial covers a few of the topics you guys had requested. You wanted to know how to move the objects you've made around the screen in 3D. You wanted to know how to draw a bitmap to the screen, without the black part of the image covering up what's behind it. You wanted simple animation and more uses for blending. This tutorial will teach you all of that. You'll notice there's no spinning boxes. The previous tutorials covered the basics of OpenGL. Each tutorial expanded on the last. This tutorial is a combination of everything that you have learned up till now, along with information on how to move your object in 3D. This tutorial is a little more advanced, so make sure you understand the previous tutorials before you jump into this tutorial.

* Loading And Moving Through A 3D World:

The tutorial you have all been waiting for! This tutorial was made by a fellow programmer named Lionel Brits. In this lesson you will learn how to load a 3D world from a data file, and move through the 3D world. The code is made using lesson 1 code, however, the tutorial web page only explains the NEW code used to load the 3D scene, and move around inside the 3D world. Download the VC++ code, and follow through it as you read the tutorial. Keys to try out are [B]lend, [F]iltering, [L]ighting (light does not move with the scene however), and Page Up/Down. I hope you enjoy Lionel's contribution to the site. When I have time I'll make the Tutorial easier to follow.

* OpenGL Flag Effect:



This tutorial code brought to you by Bosco. The same guy that created the totally cool mini demo called worthless. He enjoyed everyones reaction to his demo, and decided to go one step further and explain how he does the cool effect at the end of his demo. This tutorial builds on the code from lesson 6. By the end of the tutorial you should be able to bend fold and manipulate textures of your own. It's definitely a nice effect, and alot better than flat non moving textures. If you enjoy the tutorial, please email bosco and let him know.

Display Lists:

Want to know how to speed up you OpenGL programs? Tired of writing lots of code every time you want to put an object on the screen? If so, this tutorial is definitely for you. Learn how to use OpenGL display lists. Prebuild objects and display them on the screen with just one line of code. Speed up your programs by using precompiled objects in your programs. Stop writing the same code over and over. Let display lists do all the work for you! In this tutorial we'll build the Q-Bert pyramids using just a few lines of code thanks to display lists.



Bitmap Fonts:

I think the question I get asked most often in email is "how can I display text on the screen using OpenGL?". You could always texture map text onto your screen. Of course you have very little control over the text, and unless you're good at blending, the text usually ends up mixing with the images on the screen. If you'd like an easy way to write the text you want anywhere you want on the screen in any color you want, using any of your computers built in fonts, then this tutorial is definitely for you. Bitmaps font's are 2D scalable fonts, they can not be rotated. They always face forward.



Outline Fonts:

Bitmap fonts not good enough? Do you need control over where the fonts are on the Z axis? Do you need 3D fonts (fonts with actual depth)? Do you need wireframe fonts? If so, Outline fonts are the perfect solution. You can move them along the Z axis, and they resize. You can spin them around on an axis (something you can't do with bitmap fonts), and because proper normals are generated for each character, they can be lit up with lighting. You can build Outline fonts using any of the fonts installed on your computer. Definitely a nice font to use in games and demos.

Texture Mapped Fonts:

Hopefully my last font tutorial {grin}. This time we learn a quick and fairly nice looking way to texture map fonts,



and any other 3D object on your screen. By playing around with the code, you can create some pretty cool special effects, Everything from normal texture mapped object to sphere mapped objects. In case you don't know... Sphere mapping creates a metalic looking object that reflects anything from a pattern to a picture.



* Cool Looking Fog:

This tutorial code was generously donated to the site by Chris Aliotta. It based on the code from lesson 7, that why you're seeing the famous crate again :) It's a pretty short tutorial aimed at teaching you the art of fog. You'll learn how to use 3 different fog filters, how to change the color of the fog, and how to set how far into the screen the fog starts and how far into the screen it ends. Definitely a nice effect to know!



* 2D Texture Font:

The original version of this tutorial code was written by Giuseppe D'Agata. In this tutorial you will learn how to write any character or phrase you want to the screen using texture mapped quads. You will learn how to read one of 256 different characters from a 256x256 texture map, and finally I will show you how to place each character on the screen using pixels rather than units. Even if you're not interested in drawing 2D texture mapped characters to the screen, there is lots to learn from this tutorial. Definitely worth reading!

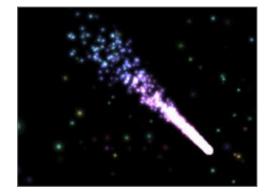


* Quadratics:

This tutorial code was written by GB Schmick the wonderful site op over at TipTup. It will introduce you to the wonderful world of quadratics. With quadratics you can easily create complex objects such as spheres, discs, cylinders and cones. These object can be created with just one line of code. With some fancy math and planning it should be possible to morph these objects from one object into another. Please let GB Schmick know what you think of the tutorial, it's always nice when visitors contribute to the site, it benefits us all. Everyone that has contributed a tutorial or project deserves credit, please let them know their work is appreciated!

Particle Engine Using Triangle Strips:

Have you ever wanted to create an explosion, water fountain, flaming star, or some other cool effect in your



OpenGL program, but writing a particle engine was either too hard, or just too complex? If so, this tutorial is for you. You'll learn how to program a simple but nice looking particle engine. I've thrown in a few extras like a rainbow mode, and lots of keyboard interaction. You'll also learn how to create OpenGL triangle strips. I hope you find the code both useful and entertaining.

Masking:



Up until now we've been blending our images onto the screen. Although this is effective, and it adds our image to the scene, a transparent object is not always pretty. Lets say you're making a game and you want solid text, or an odd shaped console to pop up. With the blending we have been using up until now, the scene will shine through our objects. By combining some fancy blending with an image mask, your text can be solid. You can also place solid oddly shaped images onto the screen. A tree with solid branches and non transparent leaves or a window, with transparent glass and a solid frame. Lots of possiblities!

Image: Control of the contro

Lines, Antialiasing, Timing, Ortho View And Simple Sounds:

This is my first large tutorial. In this tutorial you will learn about: Lines, Anti-Aliasing, Orthographic Projection, Timing, Basic Sound Effects, and Simple Game Logic. Hopefully there's enough in this tutorial to keep everyone happy :) I spent 2 days coding this tutorial, and about 2 weeks writing this HTML file. If you've ever played Amidar, the game you write in this tutorial may bring back memories. You have to fill in a grid while avoiding nasty enemies. A special item appears from time to time to help make life easier. Learn lots and have fun doing it!

* Bump-Mapping, Multi-Texturing & Extensions:

This tutorial code was written by Jens Schneider. Right off the start I'd like to point out that this is an advanced tutorial. If you're still uncertain about the basics, please go back and read the previous tutorials. If you're a new GL programmer, this lesson may be a bit much. In this lesson, you will modify the code from lesson 6 to support hardware multi-texturing on cards that support it, along with a really cool visual effect called bumpmapping. Please let Jens Schneider know what you think of the tutorial, it's always nice when visitors contribute to the site, it benefits us all. Everyone that has contributed a tutorial or project deserves credit, please let them know their work is appreciated!

* Using Direct Input With OpenGL:

This tutorial code was written by Justin Eslinger and is based on lesson 10. Instead of focusing on OpenGL this tutorial will teach you how to use DirectInput in your





OpenGL programs. I have had many requests for such a tutorial, so here it is. The code in lesson 10 will be modified to allow you to look around with the mouse and move with the arrow keys. Something you should know if you plan to write that killer 3D engine :) I hope you appreciate Justin's work. He spent alot of time making the tutorial unique (reading textures from the data file, etc), and I spent alot of time tweaking things, and making the HTML look pretty. If you enjoy this tutorial let him know!



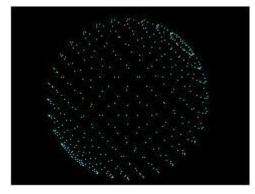
		GeForce 256/AGP/SSE NVIDIA Corporation	
	GL_NV_reg	ister_combiners	
32	GL_NV_tex	gen_emboss gen_reflection	
34	GL_NV_ver	ture_esv_combine4 tex_array_range	
		ultitexture exture_lod	
	GL_WIN_SW WGL_EXT_S	ap_hint wap_control	
		Sele Productions	



This tutorial code was written by GB Schmick and is based on his quadratics tutorial (lesson 18). In lesson 15 (texture mapped fonts) I talked a little bit about sphere mapping. I explained how to auto-generate texture coordinates, and how to set up sphere mapping, but because lesson 15 was fairly simple I decided to keep the tutorial simple, leaving out alot of details in regards to sphere mapping. Now that the tutorials are a little more advanced it's time to dive into the world of sphere mapping. TipTup did an excellent job on the tutorial, so if you appreciate his work, let him know!

Tokens, Extensions, Scissor Testing And TGA Loading:

In this tutorial I will teach you how to read and parse what OpenGL extensions are supported by your video card. I will also show you how to use scissor testing to create a cool scrolling window effect. And most importantly I will show you how to load and use TGA (targa) image files as textures in projects of your own. TGA files support the alpha channel, allowing you to create some great blending effects, and they are easy to create and work with. Not only that, by using TGA files, we no longer depend on the gIAUX library. Something I'm sure alot of you guys will appreciate!

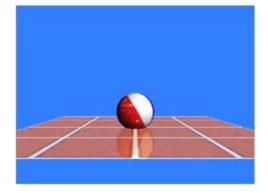


* Morphing & Loading Objects From A File:

This tutorial code was written by Piotr Cieslak. Learn how to load simple objects from a text file, and morph smoothly from one object into another. The effect in this tutorial has to be seen to be appreciated. The effect taught in this demo can be used to animated objects similar to the swimming dolphin in my Dolphin demo, or to twist and bend objects into many different shapes. You can also modify the code to use lines or solid polygons. Great effect! Hope you appreciate Piotr's work!

* Clipping & Reflections Using The Stencil Buffer:

This tutorial was written by Banu Cosmin. It demonstrates how to create extremely realistic



reflections using the stencil buffer, clipping, and multitexturing. This tutorial is more advanced than previous tutorials, so please make sure you've read the previous tutorials before jumping in. It's also important to note this tutorial will not run on video cards that do not support the stencil buffer (voodoo 1, 2, perhaps more). If you appreciate Banu's work, let him know!



* Shadows:

This is an advanced tutorial. Before you decide to try out shadows, make sure you completely understand the base code, and make sure you are familiar with the stencil buffer. This tutorial was made possible by both Banu Cosmin & Brett Porter. Banu wrote the original code. Brett cleaned the code up, combined it into one file, and wrote the HTML for the tutorial. The effect is amazing! Shadows that actual wrap around objects, and distort on the walls and floor. Thanks to Banu and Brett for their hard work, this is truely a great tutorial!



* Bezier Patches / Fullscreen Fix:

David Nikdel is the man behind this super cool tutorial. Learn how to create bezier patches. Learn how to alter a surface by modifying control points. The surface being altered is fully texture mapped, the animation is smooth! Left and Right arrow keys rotate the object while the Up and Down arrows raise and lower the resolution. This tutorial also eliminates the fullscreen problems a few of you have been having! Thanks to David for modifying the code! If you appreciate his work, let him know!

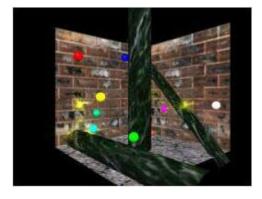


* Blitter Function, RAW Texture Loading:

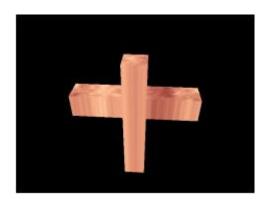
This tutorial was written by Andreas Löffler. In this tutorial you will learn how to load .RAW image files. You will also learn how to write your own blitter routine to modify textures after they have been loaded. You can copy sections of the first texture into a second texture, you can blend textures together, and you can stretch textures. The same routine can be modified to create realtime plasma and other cool effects! If you enjoy the tutorial let Andreas know!

* Collision Detection:

The tutorial you have all been waiting for. This amazing tutorial was written by Dimitrios Christopoulos. In this tutorial you will learn the basics of collision detection,



collision response, and physically based modelling effects. This tutorial concentrates more on how collision detection works than on the actual code, although all of the important code is explained. It's important to note, this is an ADVANCED tutorial. Don't expect to read through the tutorial once and understand everything about collision detection. It's a complex topic, this tutorial will get you started.



* Model Loading:

Brett Porter is the author of this tutorial. What can I say... Another incredible tutorial! This tutorial will teach you how to load in and display texture mapped Milkshape3D models. This tutorial is quite advanced so make sure you understand the previous tutorials before you attempt the code in this tutorial. It sounds as though Brett is planning a future tutorial on Skeletal Animation so if you enjoy this tutorial, show him your support! Email him and let him know you appreciate his work!

I am not a guru programmer. I am an average programmer, learning new things about OpenGL every day. I do not claim to know everything. I do not guarantee my code is bug free. I have made every effort humanly possible to eliminate all bugs but this is not always an easy task. Please keep this in mind while going through the tutorials!

Lesson 1

Welcome to my OpenGL tutorials. I am an average guy with a passion for OpenGL! The first time I heard about OpenGL was back when 3Dfx released their Hardware accelerated OpenGL driver for the Voodoo 1 card. Immediately I knew OpenGL was something I had to learn. Unfortunately, it was very hard to find any information about OpenGL in books or on the net. I spent hours trying to make code work and even more time begging people for help in email and on IRC. I found that those people that understood OpenGL considered themselves elite, and had no interest in sharing their knowledge. VERY frustrating!

I created this web site so that people interested in learning OpenGL would have a place to come if they needed help. In each of my tutorials I try to explain, in as much detail as humanly possible, what each line of code is doing. I try to keep my code simple (no MFC code to learn)! An absolute newbie to both Visual C++ and OpenGL should be able to go through the code, and have a pretty good idea of what's going on. My site is just one of many sites offering OpenGL tutorials. If you're a hardcore OpenGL programmer, my site may be too simplistic, but if you're just starting out, I feel my site has a lot to offer!

This tutorial was completely rewritten January 2000. This tutorial will teach you how to set up an OpenGL window. The window can be windowed or fullscreen, any size you want, any resolution you want, and any color depth you want. The code is very flexible and can be used for all your OpenGL projects. All my tutorials will be based on this code! I wrote the code to be flexible, and powerful at the same time. All errors are reported. There should be no memory leaks, and the code is easy to read and easy to modify. Thanks to Fredric Echols for his modifications to the code!

I'll start this tutorial by jumping right into the code. The first thing you will have to do is build a project in Visual C++. If you don't know how to do that, you should not be learning OpenGL, you should be learning Visual C++. The downloadable code is Visual C++ 6.0 code. Some versions of VC++ require that **bool** is changed to **BOOL**, **true** is changed to **TRUE**, and **false** is changed to **FALSE**. By making the changes mentioned, I have been able to compile the code on Visual C++ 4.0 and 5.0 with no other problems.

After you have created a new Win32 Application (**NOT** a console application) in Visual C++, you will need to link the OpenGL libraries. In Visual C++ go to Project, Settings, and then click on the LINK tab. Under "Object/Library Modules" at the beginning of the line (before kernel32.lib) add **OpenGL32.lib** GLu32.lib and GLaux.lib. Once you've done this click on OK. You're now ready to write an OpenGL Windows program.

The first 4 lines include the header files for each library we are using. The lines look like this:

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>

// Header

Next you need to set up all the variables you plan to use in your program. This program will create a blank OpenGL window, so we won't need to set up a lot of variables just yet. The few variables that we do set up are very important, and will be used in just about every OpenGL program you write using this code.

The first line sets up a Rendering Context. Every OpenGL program is linked to a Rendering Context. A Rendering Context is what links OpenGL calls to the Device Context. The OpenGL Rendering Context is defined as **hRC**. In order for your program to draw to a Window you need to create a Device Context, this is done in the second line. The Windows Device Context is defined as **hDC**. The DC connects the Window to the GDI (Graphics Device Interface). The RC connects OpenGL to the DC.

In the third line the variable **hWnd** will hold the handle assigned to our window by Windows, and finally, the fourth line creates an Instance (occurrence) for our program.

HDC	hDC=NULL;	11	Privat
HGLRC	hRC=NULL;	//	Perman
HWND	hWnd=NULL;		
HINSTANCEhInstan	ei	//	Holds '

The first line below sets up an array that we will use to monitor key presses on the keyboard. There are many ways to watch for key presses on the keyboard, but this is the way I do it. It's reliable, and it can handle more than one key being pressed at a time.

The **active** variable will be used to tell our program whether or not our Window has been minimized to the taskbar or not. If the Window has been minimized we can do anything from suspend the code to exit the program. I like to suspend the program. That way it won't keep running in the background when it's minimized.

The variable **fullscreen** is fairly obvious. If our program is running in fullscreen mode, **fullscreen** will be TRUE, if our program is running in Windowed mode, **fullscreen** will be FALSE. It's important to make this global so that each procedure knows if the program is running in fullscreen mode or not.

bool keys[256]; bool active=TRUE; bool fullscreen=TRUE;

Now we have to define WndProc(). The reason we have to do this is because CreateGLWindow() has a reference to WndProc() but WndProc() comes after CreateGLWindow(). In C if we want to access a procedure or section of code that comes after the section of code we are currently in we have to declare the section of code we wish to access at the top of our program. So in the following line we define WndProc() so that CreateGLWindow() can make reference to WndProc().

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// Declara

// Fullsc:

The job of the next section of code is to resize the OpenGL scene whenever the window (assuming you are using a Window rather than fullscreen mode) has been resized. Even if you are not able to resize the window (for example, you're in fullscreen mode), this routine will still be called at least once when the program is first run to set up our perspective view. The OpenGL scene will be resized based on the width and height of the window it's being displayed in.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize
{
    if (height==0)
    {
        height=1;
        // Making
    }
    glViewport(0, 0, width, height); // Reset '
```

The following lines set the screen up for a perspective view. Meaning things in the distance get smaller. This creates a realistic looking scene. The perspective is calculated with a 45 degree viewing angle based on the windows width and height. The 0.1f, 100.0f is the starting point and ending point for how deep we can draw into the screen.

glMatrixMode(GL_PROJECTION) indicates that the next 2 lines of code will affect the projection matrix. The perspective matrix is responsible for adding perspective to our scene. glLoadIdentity() is similar to a reset. It restores the selected matrix to it's original state. After glLoadIdentity() has been called we set up our perspective view for the scene. glMatrixMode(GL_MODELVIEW) indicates that any new transformations will affect the modelview matrix. The modelview matrix is where our object information is stored. Lastly we reset the modelview matrix. Don't worry if you don't understand this stuff, I will be explaining it all in later tutorials. Just know that it HAS to be done if you want a nice perspective scene.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity(); // Reset '
// Calculate The Aspect Ratio Of The Window
gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,100.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // Select
// Reset '
```

In the next section of code we do all of the setup for OpenGL. We set what color to clear the screen to, we turn on the depth buffer, enable smooth shading, etc. This routine will not be called until the OpenGL Window has been created. This procedure returns a value but because our initialization isn't that complex we wont worry about the value for now.

```
int InitGL(GLvoid)
{
```

}

The next line enables smooth shading. Smooth shading blends colors nicely across a polygon, and smoothes out lighting. I will explain smooth shading in more detail in another tutorial.

// All Se

glShadeModel(GL_SMOOTH);

The following line sets the color of the screen when it clears. If you don't know how colors work, I'll quickly explain. The color values range from 0.0f to 1.0f. 0.0f being the darkest and 1.0f being the brightest. The first parameter after glClearColor is the Red Intensity, the second parameter is for Green and the third is for Blue. The higher the number is to 1.0f, the brighter that specific color will be. The last number is an Alpha value. When it comes to clearing the screen, we wont worry about the 4th number. For now leave it at 0.0f. I will explain its use in another tutorial.

You create different colors by mixing the three primary colors for light (red, green, blue). Hope you learned primaries in school. So, if you had glClearColor(0.0f,0.0f,1.0f,0.0f) you would be clearing the screen to a bright blue. If you had glClearColor(0.5f,0.0f,0.0f,0.0f) you would be clearing the screen to a medium red. Not bright (1.0f) and not dark (0.0f). To make a white background, you would set all the colors as high as possible (1.0f). To make a black background you would set all the colors to as low as possible (0.0f).

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

The next three lines have to do with the Depth Buffer. Think of the depth buffer as layers into the screen. The depth buffer keeps track of how deep objects are into the screen. We won't really be using the depth buffer in this program, but just about every OpenGL program that draws on the screen in 3D will use the depth buffer. It sorts out which object to draw first so that a square you drew behind a circle doesn't end up on top of the circle. The depth buffer is a very important part of OpenGL.

glClearDepth(1.0f); glEnable(GL_DEPTH_TEST); glDepthFunc(GL_LEQUAL);

// Enable

Next we tell OpenGL we want the best perspective correction to be done. This causes a very tiny performance hit, but makes the perspective view look a bit better.

glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

// Really

Finally we return TRUE. If we wanted to see if initialization went ok, we could check to see if TRUE or FALSE was returned. You can add code of your own to return FALSE if an error happens. For now we won't worry about it.

return TRUE;

}

This section is where all of your drawing code will go. Anything you plan to display on the screen will go in this section of code. Each tutorial after this one will add code to this section of the program. If you already have an understanding of OpenGL, you can try creating basic shapes by adding OpenGL code below glLoadldentity() and before return TRUE. If you're new to OpenGL, wait for my next tutorial. For now all we will do is clear the screen to the color we previously decided on, clear the depth buffer and reset the scene. We wont draw anything yet.

The return TRUE tells our program that there were no problems. If you wanted the program to stop for some reason, adding a return FALSE line somewhere before return TRUE will tell our program that the drawing code failed. The program will then quit.

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    return TRUE;
}
```

The next section of code is called just before the program quits. The job of KillGLWindow() is to release the Rendering Context, the Device Context and finally the Window Handle. I've added a lot of error checking. If the program is unable to destroy any part of the Window, a message box with an error message will pop up, telling you what failed. Making it a lot easier to find problems in your code.

```
GLvoid KillGLWindow(GLvoid)
{
```

The first thing we do in KillGLWindow() is check to see if we are in fullscreen mode. If we are, we'll switch back to the desktop. We should destroy the Window before disabling fullscreen mode, but on some video cards if we destroy the Window BEFORE we disable fullscreen mode, the desktop will become corrupt. So we'll disable fullscreen mode first. This will prevent the desktop from becoming corrupt, and works well on both Nvidia and 3dfx video cards!

if (fullscreen)
{

}

We use ChangeDisplaySettings(NULL,0) to return us to our original desktop. Passing NULL as the first parameter and 0 as the second parameter forces Windows to use the values currently stored in the Windows registry (the default resolution, bit depth, frequency, etc) effectively restoring our original desktop. After we've switched back to the desktop we make the cursor visible again.

```
ChangeDisplaySettings(NULL,0);
ShowCursor(TRUE);
```

// Show M

// Proper

// Clear '

// Reset '

The code below checks to see if we have a Rendering Context (**hRC**). If we don't, the program will jump to the section of code below that checks to see if we have a Device Context.

```
if (hRC)
                                                                                           // Do We 1
      {
If we have a Rendering Context, the code below will check to see if we are able to release it (detach
the hRC from the hDC). Notice the way I'm checking for errors. I'm basically telling our program to
try freeing it (with wglMakeCurrent(NULL,NULL), then I check to see if freeing it was successful or
not. Nicely combining a few lines of code into one line.
                if (!wglMakeCurrent(NULL,NULL))
                 {
If we were unable to release the DC and RC contexts, MessageBox() will pop up an error message
letting us know the DC and RC could not be released. NULL means the message box has no
parent Window. The text right after NULL is the text that appears in the message box.
"SHUTDOWN ERROR" is the text that appears at the top of the message box (title). Next we have
MB_OK, this means we want a message box with one button labelled "OK".
MB_ICONINFORMATION makes a lower case i in a circle appear inside the message box (makes
it stand out a bit more).
                           MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR",
                 }
Next we try to delete the Rendering Context. If we were unsuccessful an error message will pop up.
                if (!wglDeleteContext(hRC))
                                                                                           // Are We
                 {
If we were unable to delete the Rendering Context the code below will pop up a message box letting
us know that deleting the RC was unsuccessful. hRC will be set to NULL.
```

```
MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN ER
}
hRC=NULL; // Set RC
}
```

Now we check to see if our program has a Device Context and if it does, we try to release it. If we're unable to release the Device Context an error message will pop up and **hDC** will be set to NULL.

```
if (hDC && !ReleaseDC(hWnd,hDC)) // Are We
{
    MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",MB_OK |
    hDC=NULL; // Set DC
}
```

Now we check to see if there is a Window Handle and if there is, we try to destroy the Window using DestroyWindow(**hWnd**). If we are unable to destroy the Window, an error message will pop up and **hWnd** will be set to NULL.

Last thing to do is unregister our Windows Class. This allows us to properly kill the window, and then reopen another window without receiving the error message "Windows Class already registered".

The next section of code creates our OpenGL Window. I spent a lot of time trying to decide if I should create a fixed fullscreen Window that doesn't require a lot of extra code, or an easy to customize user friendly Window that requires a lot more code. I decided the user friendly Window with a lot more code would be the best choice. I get asked the following questions all the time in email: How can I create a Window instead of using fullscreen? How do I change the Window's title? How do I change the resolution or pixel format of the Window? The following code does all of that! Therefore it's better learning material and will make writing OpenGL programs of your own a lot easier!

As you can see the procedure returns BOOL (TRUE or FALSE), it also takes 5 parameters: **title** of the Window, **width** of the Window, **height** of the Window, **bits** (16/24/32), and finally **fullscreenflag** TRUE for fullscreen or FALSE for windowed. We return a boolean value that will tell us if the Window was created successfully.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag) {
```

When we ask Windows to find us a pixel format that matches the one we want, the number of the mode that Windows ends up finding for us will be stored in the variable **PixelFormat**.

GLuint

}

PixelFormat;

wc will be used to hold our Window Class structure. The Window Class structure holds information about our window. By changing different fields in the Class we can change how the window looks and behaves. Every window belongs to a Window Class. Before you create a window, you MUST register a Class for the window.

WNDCLASS wc;

dwExStyle and **dwStyle** will store the Extended and normal Window Style Information. I use variables to store the styles so that I can change the styles depending on what type of window I need to create (A popup window for fullscreen or a window with a border for windowed mode)

DWORD DWORD dwExStyle; dwStyle;

The following 5 lines of code grab the upper left, and lower right values of a rectangle. We'll use these values to adjust our window so that the area we draw on is the exact resolution we want. Normally if we create a 640x480 window, the borders of the window take up some of our resolution.

RECT WindowRect; WindowRect.left=(long)0; WindowRect.right=(long)width; WindowRect.top=(long)0; WindowRect.bottom=(long)height;

In the next line of code we make the global variable **fullscreen** equal **fullscreenflag**. So if we made our Window fullscreen, the variable **fullscreenflag** would be TRUE. If we didn't make the variable **fullscreen** equal **fullscreenflag**, the variable **fullscreen** would stay FALSE. If we were killing the window, and the computer was in fullscreen mode, but the variable fullscreen was FALSE instead of TRUE like it should be, the computer wouldn't switch back to the desktop, because it would think it was already showing the desktop. God I hope that makes sense. Basically to sum it up, **fullscreen** has to equal whatever **fullscreenflag** equals, otherwise there will be problems.

fullscreen=fullscreenflag;

In the next section of code, we grab an instance for our Window, then we define the Window Class.

The style CS_HREDRAW and CS_VREDRAW force the Window to redraw whenever it is resized. CS_OWNDC creates a private DC for the Window. Meaning the DC is not shared across applications. WndProc is the procedure that watches for messages in our program. No extra Window data is used so we zero the two fields. Then we set the instance. Next we set hIcon to NULL meaning we don't want an ICON in the Window, and for a mouse pointer we use the standard arrow. The background color doesn't matter (we set that in GL). We don't want a menu in this Window so we set it to NULL, and the class name can be any name you want. I'll use "OpenGL" for simplicity.



// Grabs 1 // Set Le:

// Window

// Set Th

hInstance	= GetModuleHandle(NULL);	// Grab An Instance
wc.style	= CS_HREDRAW CS_VREDRAW CS_OWNDC;	// Redraw
wc.lpfnWndProc	= (WNDPROC) WndProc;	
wc.cbClsExtra	= 0;	
wc.cbWndExtra	= 0;	
wc.hInstance	= hInstance;	
wc.hIcon	= LoadIcon(NULL, IDI_WINLOGO);	// Load T
wc.hCursor	= LoadCursor(NULL, IDC_ARROW);	
wc.hbrBackground	= NULL;	// No Bacl
wc.lpszMenuName	= NULL;	
wc.lpszClassName	= "OpenGL";	// Set Th

Now we register the Class. If anything goes wrong, an error message will pop up. Clicking on OK in the error box will exit the program.

```
if (!RegisterClass(&wc)) // Attemp
{
    MessageBox(NULL,"Failed To Register The Window Class.","ERROR",MB_OK|MB_I'
    return FALSE;
}
```

Now we check to see if the program should run in fullscreen mode or windowed mode. If it should be fullscreen mode, we'll attempt to set fullscreen mode.

```
if (fullscreen)
{
```

The next section of code is something people seem to have a lot of problems with... switching to fullscreen mode. There are a few very important things you should keep in mind when switching to full screen mode. Make sure the width and height that you use in fullscreen mode is the same as the width and height you plan to use for your window, and most importantly, set fullscreen mode BEFORE you create your window. In this code, you don't have to worry about the width and height, the fullscreen and the window size are both set to be the size requested.

DEVMODE dmScreenSettings;		// Device		
<pre>memset(&dmScreenSettings,0,sizeof(dmScreenSettings));</pre>				
dmScreenSettings.dmSize=sizeof(dmS	ScreenSettings);	// Size O		
dmScreenSettings.dmPelsWidth	= width;	// Select		
dmScreenSettings.dmPelsHeight	= height;	// Select		
dmScreenSettings.dmBitsPerPel	= bits;			
dmScreenSettings.dmFields=DM BITSP	PERPEL DM_PELSWIDTH DM_PEI	LSHEIGHT;		

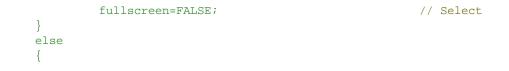
In the code above we clear room to store our video settings. We set the width, height and bits that we want the screen to switch to. In the code below we try to set the requested full screen mode. We stored all the information about the width, height and bits in dmScreenSettings. In the line below ChangeDisplaySettings tries to switch to a mode that matches what we stored in dmScreenSettings. I use the parameter CDS_FULLSCREEN when switching modes, because it's supposed to remove the start bar at the bottom of the screen, plus it doesn't move or resize the windows on your desktop when you switch to fullscreen mode and back.

// Try To Set Selected Mode And Get Results. NOTE: CDS_FULLSCREEN Gets R
if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_{
{

If the mode couldn't be set the code below will run. If a matching fullscreen mode doesn't exist, a messagebox will pop up offering two options... The option to run in a window or the option to quit.

// If The Mode Fails, Offer Two Options. Quit Or Run In A Windo
if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Suppor
{

If the user decided to use windowed mode, the variable **fullscreen** becomes FALSE, and the program continues running.



If the user decided to quit, a messagebox will pop up telling the user that the program is about to close. FALSE will be returned telling our program that the window was not created successfully. The program will then quit.

```
// Pop Up A Message Box Letting User Know The Program I
MessageBox(NULL,"Program Will Now Close.","ERROR",MB_OK
return FALSE;
}
}
```

Because the fullscreen code above may have failed and the user may have decided to run the program in a window instead, we check once again to see if **fullscreen** is TRUE or FALSE before we set up the screen / window type.

```
if (fullscreen)
{
```

If we are still in fullscreen mode we'll set the extended style to WS_EX_APPWINDOW, which force a top level window down to the taskbar once our window is visible. For the window style we'll create a WS_POPUP window. This type of window has no border around it, making it perfect for fullscreen mode.

Finally, we disable the mouse pointer. If your program is not interactive, it's usually nice to disable the mouse pointer when in fullscreen mode. It's up to you though.

```
dwExStyle=WS_EX_APPWINDOW;
dwStyle=WS_POPUP;
ShowCursor(FALSE);
}
else
{
```

// Window
// Window
// Hide Mu

If we're using a window instead of fullscreen mode, we'll add WS_EX_WINDOWEDGE to the extended style. This gives the window a more 3D look. For style we'll use WS_OVERLAPPEDWINDOW instead of WS_POPUP. WS_OVERLAPPEDWINDOW creates a window with a title bar, sizing border, window menu, and minimize / maximize buttons.

```
dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // Window
dwStyle=WS_OVERLAPPEDWINDOW;
}
```

The line below adjust our window depending on what style of window we are creating. The adjustment will make our window exactly the resolution we request. Normally the borders will overlap parts of our window. By using the AdjustWindowRectEx command none of our OpenGL scene will be covered up by the borders, instead, the window will be made larger to account for the pixels needed to draw the window border. In fullscreen mode, this command has no effect.

```
AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle); // Adjust
```

In the next section of code, we're going to create our window and check to see if it was created properly. We pass CreateWindowEx() all the parameters it requires. The extended style we decided to use. The class name (which has to be the same as the name you used when you registered the Window Class). The window title. The window style. The top left position of your window (0,0 is a safe bet). The width and height of the window. We don't want a parent window, and we don't want a menu so we set both these parameters to NULL. We pass our window instance, and finally we NULL the last parameter.

Notice we include the styles WS_CLIPSIBLINGS and WS_CLIPCHILDREN along with the style of window we've decided to use. WS_CLIPSIBLINGS and WS_CLIPCHILDREN are both REQUIRED for OpenGL to work properly. These styles prevent other windows from drawing over or into our OpenGL Window.

if (!(hWnd=CreateWindowEx(dwExStyle, "OpenGL",	// Extend // Class]
	title,	// СТАББ !
	WS_CLIPSIBLINGS	// Require
	WS_CLIPCHILDREN	// Require
	dwStyle,	// Select
	0, 0,	
	WindowRect.right-WindowRect.left,	// Calcula
	WindowRect.bottom-WindowRect.top,	// Calcula
	NULL,	
	NULL,	
	hInstance,	
	NULL)))	

Next we check to see if our window was created properly. If our window was created, **hWnd** will hold the window handle. If the window wasn't created the code below will pop up an error message and the program will quit.

```
{
    KillGLWindow();
    MessageBox(NULL,"Window Creation Error.","ERROR",MB_OK|MB_ICONEXCLAMATION
    return FALSE;
}
```

The next section of code describes a Pixel Format. We choose a format that supports OpenGL and double buffering, along with RGBA (red, green, blue, alpha channel). We try to find a pixel format that matches the bits we decided on (16bit,24bit,32bit). Finally we set up a 16bit Z-Buffer. The remaining parameters are either not used or are not important (aside from the stencil buffer and the (slow) accumulation buffer).

```
static PIXELFORMATDESCRIPTOR pfd=
                                                                           // pfd Te
{
         sizeof(PIXELFORMATDESCRIPTOR),
         1.
         PFD_DRAW_TO_WINDOW
         PFD_SUPPORT_OPENGL
         PFD_DOUBLEBUFFER,
                                                                           // Must S
         PFD_TYPE_RGBA,
         bits,
         0, 0, 0, 0, 0, 0,
                                                                           // Color 1
         Ο,
         Ο,
         0,
         0, 0, 0, 0,
         16,
         Ο,
         0,
         PFD MAIN PLANE,
         Ο,
         0, 0, 0
};
```

If there were no errors while creating the window, we'll attempt to get an OpenGL Device Context. If we can't get a DC an error message will pop onto the screen, and the program will quit (return FALSE).

```
if (!(hDC=GetDC(hWnd)))
{
            KillGLWindow();
            MessageBox(NULL,"Can't Create A GL Device Context.","ERROR",MB_OK|MB_ICON.
            return FALSE;
}
```

If we managed to get a Device Context for our OpenGL window we'll try to find a pixel format that matches the one we described above. If Windows can't find a matching pixel format, an error message will pop onto the screen and the program will quit (return FALSE).

If windows found a matching pixel format we'll try setting the pixel format. If the pixel format cannot be set, an error message will pop up on the screen and the program will quit (return FALSE).

```
if(!SetPixelFormat(hDC,PixelFormat,&pfd)) // Are We
{
        KillGLWindow();
        MessageBox(NULL,"Can't Set The PixelFormat.","ERROR",MB_OK|MB_ICONEXCLAMA'
        return FALSE;
}
```

If the pixel format was set properly we'll try to get a Rendering Context. If we can't get a Rendering Context an error message will be displayed on the screen and the program will quit (return FALSE).

If there have been no errors so far, and we've managed to create both a Device Context and a Rendering Context all we have to do now is make the Rendering Context active. If we can't make the Rendering Context active an error message will pop up on the screen and the program will quit (return FALSE).

If everything went smoothly, and our OpenGL window was created we'll show the window, set it to be the foreground window (giving it more priority) and then set the focus to that window. Then we'll call ReSizeGLScene passing the screen width and height to set up our perspective OpenGL screen.

ShowWindow(hWnd,SW_SHOW); SetForegroundWindow(hWnd); SetFocus(hWnd); ReSizeGLScene(width, height); // Show The // Slight

Finally we jump to InitGL() where we can set up lighting, textures, and anything else that needs to be setup. You can do your own error checking in InitGL(), and pass back TRUE (everythings OK) or FALSE (somethings not right). For example, if you were loading textures in InitGL() and had an error, you may want the program to stop. If you send back FALSE from InitGL() the lines of code below will see the FALSE as an error message and the program will quit.

```
if (!InitGL())
{
     KillGLWindow();
     MessageBox(NULL,"Initialization Failed.","ERROR",MB_OK|MB_ICONEXCLAMATION
     return FALSE;
}
```

If we've made it this far, it's safe to assume the window creation was successful. We return TRUE to WinMain() telling WinMain() there were no errors. This prevents the program from quitting.

return TRUE;

}

This is where all the window messages are dealt with. When we registred the Window Class we told it to jump to this section of code to deal with window messages.

LRESULT CALLBACK WndProc(HWND	hWnd,		// Handle
		UINT	uMsg,	
		WPARAM	wParam,	
		LPARAM	lParam)	
·				

{

The code below sets **uMsg** as the value that all the case statements will be compared to. **uMsg** will hold the name of the message we want to deal with.

switch (uMsg)
{

if **uMsg** is WM_ACTIVE we check to see if our window is still active. If our window has been minimized the variable **active** will be FALSE. If our window is active, the variable **active** will be TRUE.

```
case WM_ACTIVATE: // Watch :
{
    if (!HIWORD(wParam))
    {
        active=TRUE;
    }
    else
    {
        active=FALSE;
    }
    return 0; // Return
}
```

If the message is WM_SYSCOMMAND (system command) we'll compare **wParam** against the case statements. If **wParam** is SC_SCREENSAVE or SC_MONITORPOWER either a screensaver is trying to start or the monitor is trying to enter power saving mode. By returning 0 we prevent both those things from happening.

```
case WM_SYSCOMMAND:
{
    switch (wParam)
    {
        case SC_SCREENSAVE:
        case SC_MONITORPOWER:
        return 0; // Preven
    }
    break;
}
```

If **uMsg** is WM_CLOSE the window has been closed. We send out a quit message that the main loop will intercept. The variable **done** will be set to TRUE, the main loop in WinMain() will stop, and the program will close.

// Jump B

If a key is being held down we can find out what key it is by reading **wParam**. I then make that keys cell in the array **keys[]** become TRUE. That way I can read the array later on and find out which keys are being held down. This allows more than one key to be pressed at the same time.

{

}

```
keys[wParam] = TRUE;
return 0;
```

// Jump Bi

// Jump Bi

If a key has been released we find out which key it was by reading **wParam**. We then make that keys cell in the array **keys[]** equal FALSE. That way when I read the cell for that key I'll know if it's still being held down or if it's been released. Each key on the keyboard can be represented by a number from 0-255. When I press the key that represents the number 40 for example, **keys[40]** will become TRUE. When I let go, it will become FALSE. This is how we use cells to store keypresses.

```
case WM_KEYUP:
{
    keys[wParam] = FALSE;
    return 0;
}
```

Whenever we resize our window **uMsg** will eventually become the message WM_SIZE. We read the LOWORD and HIWORD values of **IParam** to find out the windows new width and height. We pass the new width and height to ReSizeGLScene(). The OpenGL Scene is then resized to the new width and height.

Any messages that we don't care about will be passed to DefWindowProc so that Windows can deal with them.

// Pass All Unhandled Messages To DefWindowProc
return DefWindowProc(hWnd,uMsg,wParam,lParam);

```
This is the entry point of our Windows Application. This is where we call our window creation routine, deal with window messages, and watch for human interaction.
```

int WINAPI WinMain(HINSTANCEhInstance,		// Instan
	HINSTANCEhPrevIn	stance,	// Previo
	LPSTR	lpCmdLine,	
	int	nCmdShow)	// Window
·			

{

}

We set up two variables. **msg** will be used to check if there are any waiting messages that need to be dealt with. the variable **done** starts out being FALSE. This means our program is not done running. As long as **done** remains FALSE, the program will continue to run. As soon as **done** is changed from FALSE to TRUE, our program will quit.

MSG msg; BOOL done=FALSE;

This section of code is completely optional. It pops up a messagebox that asks if you would like to run the program in fullscreen mode. If the user clicks on the NO button, the variable **fullscreen** changes from TRUE (it's default) to FALSE and the program runs in windowed mode instead of fullscreen mode.

```
// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start FullScreen"
{
    fullscreen=FALSE; // Window
}
```

This is how we create our OpenGL window. We pass the title, the width, the height, the color depth, and TRUE (fullscreen) or FALSE (window mode) to CreateGLWindow. That's it! I'm pretty happy with the simplicity of this code. If the window was not created for some reason, FALSE will be returned and our program will immediately quit (return 0).

```
// Create Our OpenGL Window
if (!CreateGLWindow("NeHe's OpenGL Framework",640,480,16,fullscreen))
{
    return 0; // Quit I:
}
```

This is the start of our loop. As long as **done** equals FALSE the loop will keep repeating.

```
while(!done)
{
```

The first thing we have to do is check to see if any window messages are waiting. By using PeekMessage() we can check for messages without halting our program. A lot of programs use GetMessage(). It works fine, but with GetMessage() your program doesn't do anything until it receives a paint message or some other window message.

```
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is The:
{
```

In the next section of code we check to see if a quit message was issued. If the current message is a WM_QUIT message caused by PostQuitMessage(0) the variable **done** is set to TRUE, causing the program to quit.

If the message isn't a quit message we translate the message then dispatch the message so that WndProc() or Windows can deal with it.



If there were no messages we'll draw our OpenGL scene. The first line of code below checks to see if the window is active. The scene is rendered and the returned value is checked. If DrawGLScene() returns FALSE or the ESC key is pressed the variable **done** is set to TRUE, causing the program to quit.

If everything rendered fine, we swap the buffer (By using double buffering we get smooth flicker free animation). By using double buffering, we are drawing everything to a hidden screen that we can not see. When we swap the buffer, the screen we see becomes the hidden screen, and the screen that was hidden becomes visible. This way we don't see our scene being drawn out. It just instantly appears.

```
SwapBuffers(hDC); // Swap B<sup>.</sup>
}
```

The next bit of code is new and has been added just recently (05-01-00). It allows us to press the F1 key to switch from fullscreen mode to windowed mode or windowed mode to fullscreen mode.

// Have W

```
// Is F1 1
                  if (keys[VK_F1])
                   {
                            keys[VK_F1]=FALSE;
                                                                            // If So 1
                            KillGLWindow();
                            fullscreen=!fullscreen;
                            // Recreate Our OpenGL Window
                            if (!CreateGLWindow("NeHe's OpenGL Framework",640,480,1
                            {
                                     return 0;
                                                                           // Quit I:
                            }
                  }
         }
}
```

If the **done** variable is no longer FALSE, the program quits. We kill the OpenGL window properly so that everything is freed up, and we exit the program.

```
// Shutdown
KillGLWindow();
return (msg.wParam);
```

}

In this tutorial I have tried to explain in as much detail, every step involved in setting up, and creating a fullscreen OpenGL program of your own, that will exit when the ESC key is pressed and monitor if the window is active or not. I've spent roughly 2 weeks writing the code, one week fixing bugs & talking with programming gurus, and 2 days (roughly 22 hours writing this HTML file). If you have comments or questions please email me. If you feel I have incorrectly commented something or that the code could be done better in some sections, please let me know. I want to make the best OpenGL tutorials I can and I'm interested in hearing your feedback.

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Peter De Jaegher)
- * DOWNLOAD ASM Code For This Lesson. (Conversion by Foolman)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

OpenGL On MacOS

So you've been wanting to setup OpenGL on MacOS? Here's the place to learn what you need and how you need to do it.

What You'll Need:

First and foremost, you'll need a compiler. By far the best and most popular on the Macintosh is Metrowerks Codewarrior. If you're a student, get the educational version - there's no difference between it and the professional version and it'll cost you a lot less.

Next, you'll need the OpenGL SDK (that's **S**oftware **D**evelopment **K**it) from Apple. Now we're ready to create an OpenGL program!

Getting Started with GLUT:

Ok, here is the beginning of the program, where we include headers:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include "tk.h"
```

The first is the standard OpenGL calls, the other three provide additional calls which we will use in our programs.

Next, we define some constants:

#define	k₩indow₩idth	400
#define	kWindowHeight	300

We use these for the height and width of our window. Next, the function prototypes:

```
GLvoid InitGL(GLvoid);
GLvoid DrawGLScene(GLvoid);
GLvoid ReSizeGLScene(int Width, int Height);
```

... and the main() function:

```
int main(int argc, char** argv)
{
```

```
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize (kWindowWidth, kWindowHeight);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
InitGL();
glutDisplayFunc(DrawGLScene);
glutReshapeFunc(ReSizeGLScene);
glutMainLoop();
return 0;
}
```

glutInit(), glutInitDisplayMode(), glutInitWindowSize(), glutInitWindowPosition(), and glutCreateWindow() all set up our OpenGL program. InitGL() does the same thing in the Mac program as in the Windows program. glutDisplayFunc(DrawGLScene) tells GLUT that we want the DrawGLScene function to be used when we want to draw the scene. glutReshapeFunc (ReSizeGLScene) tells GLUT that we want the ReSizeGLScene function to be used if the window is resized.

Later, we will use glutKeyboardFunc(), which tells GLUT which function we want to use when a key is pressed, and glutIdleFunc() which tells GLUT which function it will call repeatedly (we'll use it to spin stuff in space).

Finally, glutMainLoop() starts the program. Once this is called, it will only return to the main() function when the program is quitting.

You're done!

Well, that's about it. Most everything else is the same as NeHe's examples. I suggest you look at the Read Me included with the MacOS ports, as it has more detail on specific changes from the examples themselves.

Have fun!

Tony Parker, asp@usc.edu

Back To NeHe Productions!

OpenGL Under Solaris

This document describes (quick and dirty) how to install OpenGL and GLUT libraries under Solaris 7 on a Sun workstation.

The Development Tools:

Make sure you have a Solaris DEVELOPER installation on your machine. This means you have all the header files that are nessesary for program development under Solaris installed. The easiest way is to install Solaris as a development version. This can be done from the normal Solaris installation CD ROM.

After you've done this you should have your /usr/include and /usr/openwin/include directories filled with nice liddle header files.

The C Compiler:

Sun doesn't ship a C or C++ compiler with Solaris. But you're lucky. You don't have to pay :-)

http://www.sunfreeware.com/

There you find gcc the GNU Compiler Collection for Solaris precompiled and ready for easy installation. Get the version you like and install it.

> pkgadd gcc-xxxversion

This will install gcc under /usr/local. You can also do this with admintool:

> admintool

Browse->Software Edit->Add

Then choose Source: "Hard disk" and specify the directory that you've stored the package in.

I recommend also downloading and installation of the libstdc++ library if nessesary for you gcc version.

The OpenGL library

OpenGL should be shipped with Solaris these days. Check if you've already installed it.

```
> cd /usr/openwin/lib
```

```
> ls libGL*
```

This should print:

```
libGL.so@ libGLU.so@ libGLw.so@
libGL.so.1* libGLU.so.1* libGLw.so.1*
```

This means that you have the libraries already installed (runtime version).

But are the header files also there?

> cd /usr/openwin/include/GL

> ls

This should print:

gl.h	glu.h	glxmd.h	glxtokens.h
glmacros.h	glx.h	glxproto.h	

I have it. But what version is it?

This is a FAQ.

http://www.sun.com/software/graphics/OpenGL/Developer/FAQ-1.1.2.html

Helps you with questions dealing with OpenGL on Sun platforms.

Yes cool. Seems they're ready. Skip the rest of this step and go to GLUT.

You don't already have OpenGL? Your version is too old? Download a new one:

http://www.sun.com/solaris/opengl/

Helps you. Make sure to get the nessesary patches for your OS version and install them. BTW. You need root access to do this. Ask you local sysadmin to do it for you. Follow the online guide for installation.

GLUT

Now you have OpenGL but not GLUT. Where can you get it? Look right here:

http://www.sun.com/software/graphics/OpenGL/Demos/index.html

Following the links will take you to this location:

http://reality.sgi.com/opengl/glut3/glut3.html#sun

I've personally downloaded the 32bit version unless I run the 64 bit kernel of Solaris. I've installed GLUT under /usr/local. This is normally a good place for stuff like this.

Well I have it, but when I try to run the samples in progs/ it claims that it can't find libglut.a. To tell your OS where to look for runtime libraries you need to add the path to GLUT to your variable LD_LIBRARY_PATH.

If you're using /bin/sh do something like this:

> LD_LIBRARY_PATH=/lib:/usr/lib:/usr/openwin/lib:/usr/dt/lib:/usr/local/lib:/usr/local/spare

> export LD_LIBRARY_PATH

If you're using a csh do something like this:

>setenv LD_LIBRARY_PATH /lib:/usr/lib:/usr/openwin/lib:/usr/dt/lib:/usr/local/lib:/usr/local

Verify that everything is correct:

> echo \$LD_LIBRARY_PATH

/lib:/usr/lib:/usr/openwin/lib:/usr/dt/lib:/usr/local/lib:/usr/local/sparc_solaris/glut-3.7,

Congratulations you're done!

That's it folks. Now you should be ready to compile and run NeHe's OpenGL tutorials.

If you find spelling mistakes (I'm not a native english speaking beeing), errors in my description, outdated links, or have a better install procedure please contact me.

- Lakmal Gunasekara 1999 for NeHe Productions.

Back To NeHe Productions!

OpenGL On MacOS X Public Beta

So you've been wanting to setup OpenGL on MacOS X? Here's the place to learn what you need and how you need to do it. This is a direct port from the MacOS ports, so if something seems familiar, thats why ;)

What You'll Need:

You will need a compiler. Two compilers are currently available, Apple's "Project Builder" and Metrowerks CodeWarrior. Project Builder is being made free in Mid-October(2000), so this tutorial will demonstrate how to make a GLUT project in Project Builder.

Getting Started with Project Builder:

This bit is easy. Just choose "File->New Project" and select a "Cocoa Application." Now choose the name of your project, and your project IDE will pop up.

Now goto the "Project" Menu and "Add Framework..." to add the GLUT.framework

Getting Started with GLUT:

Ok, here is the beginning of the program, where we include headers, notice there is only one header, as opposed to three:

#include <GLUT/glut.h>

The first is the standard OpenGL calls, the other three provide additional calls which we will use in our programs.

Next, we define some constants:

#define kWindowWidth400#define kWindowHeight300

We use these for the height and width of our window. Next, the function prototypes:

```
GLvoid InitGL(GLvoid);
GLvoid DrawGLScene(GLvoid);
GLvoid ReSizeGLScene(int Width, int Height);
```

... and the main() function:

Jeff Molofee's OpenGL Mac OS X Tutorial (By Raal Goff)

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (kWindowWidth, kWindowHeight);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    InitGL();
    glutDisplayFunc(DrawGLScene);
    glutReshapeFunc(ReSizeGLScene);
    glutMainLoop();
    return 0;
}
```

glutInit(), glutInitDisplayMode(), glutInitWindowSize(), glutInitWindowPosition(), and glutCreateWindow() all set up our OpenGL program. InitGL() does the same thing in the Mac program as in the Windows program. glutDisplayFunc(DrawGLScene) tells GLUT that we want the DrawGLScene function to be used when we want to draw the scene. glutReshapeFunc (ReSizeGLScene) tells GLUT that we want the ReSizeGLScene function to be used if the window is resized.

Later, we will use glutKeyboardFunc(), which tells GLUT which function we want to use when a key is pressed, and glutIdleFunc() which tells GLUT which function it will call repeatedly (we'll use it to spin stuff in space).

Finally, glutMainLoop() starts the program. Once this is called, it will only return to the main() function when the program is quitting.

All Done!

Notice the only real difference here is that we are changing the headers. Pretty simple!

In later tutorials there will be some bigger differences, but for now its just as simple as changing the headers and adding the framework.

Have fun!

R.Goff (unreality@mac.com)

Lesson 2

In the first tutorial I taught you how to create an OpenGL Window. In this tutorial I will teach you how to create both Triangles and Quads. We will create a triangle using GL_TRIANGLES, and a square using GL_QUADS.

Using the code from the first tutorial, we will be adding to the DrawGLScene() procedure. I will rewrite the entire procedure below. If you plan to modify the last lesson, you can replace the DrawGLScene() procedure with the code below, or just add the lines of code below that do not exist in the last tutorial.

int DrawGLScene(GLvoid) {	// Here's
glClear(GL_COLOR_BUFFER_BIT GL_DEPTH_BUFFER_BIT);	// Clear The Scree
glLoadIdentity();	// Reset The View

When you do a glLoadIdentity() what you are doing is moving back to the center of the screen with the X axis running left to right, the Y axis moving up and down, and the Z axis moving into, and out of the screen. The center of an OpenGL screen is 0.0f on the X and Y axis. To the left of center would be a negative number. To the right would be a positive number. Moving towards the top of the screen would be a positive number, moving to the bottom of the screen would be a negative number. Moving deeper into the screen is a negative number, moving towards the viewer would be a positive number.

glTranslatef(x, y, z) moves along the X, Y and Z axis, in that order. The line of code below moves left on the X axis 1.5 units. It does not move on the Y axis at all (0.0), and it moves into the screen 6.0 units. When you translate, you are not moving a set amount from the center of the screen, you are moving a set amount from wherever you currently were on the screen.

glTranslatef(-1.5f,0.0f,-6.0f);

Now that we have moved to the left half of the screen, and we've set the view deep enough into the screen (6.0) that we can see our entire scene we will create the Triangle. glBegin(GL_TRIANGLES) means we want to start drawing a triangle, and glEnd() tells OpenGL we are done creating the triangle. Typically if you want 3 points, use GL_TRIANGLES. Drawing triangles is fairly fast on most video cards. If you want 4 points use GL_QUADS to make life easier. From what I've heard, most video cards just draw to triangles anyways. Finally if you want more than 4 points, use GL_POLYGON.

In our simple program, we draw just one triangle. If we wanted to draw a second triangle, we could include another 3 lines of code (3 points) right after the first three. All six lines of code would be between glBegin(GL_TRIANGLES) and glEnd(). There's no point in putting a glBegin (GL_TRIANGLES) and a glEnd() around every group of 3 points if we're drawing all triangles. This applies to quads as well. If you know you're drawing all quads, you can include the second four lines of code right after the first four lines. A polygon on the other hand (GL_POLYGON) can be made up of any amount of point so it doesn't matter how many lines you have between glBegin (GL_POLYGON) and glEnd().

// Move L

The first line after glBegin, sets the first point of our polygon. The first number of glVertex is for the X axis, the second number is for the Y axis, and the third number is for the Z axis. So in the first line, we don't move on the X axis. We move up one unit on the Y axis, and we don't move on the Z axis. This gives us the top point of the triangle. The second glVertex moves left one unit on the X axis and down one unit on the Y axis. This gives us the bottom left point of the triangle. The third glVertex moves right one unit, and down one unit. This gives us the bottom right point of the triangle. glEnd() tells OpenGL there are no more points. The filled triangle will be displayed.

glBegin(GL_TRIANGLES); glVertex3f(0.0f, 1.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f); glVertex3f(1.0f,-1.0f, 0.0f); glEnd();

Now that we have the triangle displayed on the left half of the screen, we need to move to the right half of the screen to display the square. In order to do this we use glTranslate again. This time we must move to the right, so X must be a positive value. Because we've already moved left 1.5 units, to get to the center we have to move right 1.5 units. After we reach the center we have to move another 1.5 units to the right of center. So in total we need to move 3.0 units to the right.

glTranslatef(3.0f,0.0f,0.0f);

}

Now we create the square. We'll do this using GL_QUADS. A quad is basically a 4 sided polygon. Perfect for making a square. The code for creating a square is very similar to the code we used to create a triangle. The only difference is the use of GL QUADS instead of GL TRIANGLES, and an extra glVertex3f for the 4th point of the square. We'll draw the square top left, top right, bottom right, bottom left.

glBegin(GL_QUADS);	// Draw A Quad
glVertex3f(-1.0f, 1.0f, 0.0f);	// Top Le:
glVertex3f(1.0f, 1.0f, 0.0f);	// Top Rig
glVertex3f(1.0f,-1.0f, 0.0f);	// Bottom
glVertex3f(-1.0f,-1.0f, 0.0f);	// Bottom
glEnd();	// Done Drawing The
return TRUE;	// Keep G

Finally change the code to toggle window / fullscreen mode so that the title at the top of the window is proper.

if {	(keys[VK_F1])	// Is Fl Being Pre:
	keys[VK_F1]=FALSE;	// If So Make Key 1
	KillGLWindow();	// Kill O
	fullscreen=!fullscreen;	// Toggle
	// Recreate Our OpenGL Window	(Modified)
	if (!CreateGLWindow("NeHe's Fi	irst Polygon Tutorial",640
	{	
	return 0;	// Quit If Window 1
	}	

Page 2 of 3

// Move R

// Drawing

// Bottom

// Bottom

// Top

// Finished Drawing

}

In this tutorial I have tried to explain in as much detail, every step involved in drawing polygons, and quads on the screen using OpenGL. If you have comments or questions please email me. If you feel I have incorrectly commented something or that the code could be done better in some sections, please let me know. I want to make the best OpenGL tutorials I can. I'm interested in hearing your feedback.

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Peter De Jaegher)
- * DOWNLOAD ASM Code For This Lesson. (Conversion by Foolman)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 3

In the last tutorial I taught you how to display Triangles and Quads on the screen. In this tutorial I will teach you how to add 2 different types of coloring to the triangle and quad. Flat coloring will make the quad one solid color. Smooth coloring will blend the 3 colors specified at each point (vertex) of the triangle together, creating a nice blend of colors.

Using the code from the last tutorial, we will be adding to the DrawGLScene procedure. I will rewrite the entire procedure below, so if you plan to modify the last lesson, you can replace the DrawGLScene procedure with the code below, or just add code to the DrawGLScene procedure that is not already in the last tutorial.

```
int DrawGLScene(GLvoid)
                                                                         // Here's Where We
{
        glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT); // Clear The Screen And The
        glLoadIdentity();
                                                               // Reset The Current Modelvi
                                                                        // Left 1.5 Then I
        glTranslatef(-1.5f,0.0f,-6.0f);
        glBegin(GL_TRIANGLES);
                                                                        // Begin Drawing T
```

If you remember from the last tutorial, this is the section of code to draw the triangle on the left half of the screen. The next line of code will be the first time we use the command glColor3f(r,g,b). The three parameters in the brackets are red, green and blue intensity values. The values can be from 0.0f to 1.0f. It works the same way as the color values we use to clear the background of the screen.

We are setting the color to red (full red intensity, no green, no blue). The line of code right after that is the first vertex (the top of the triangle), and will be drawn using the current color which is red. Anything we draw from now on will be red until we change the color to something other than red.

glColor3f(1.0f,0.0f,0.0f); // Set The Color To Red glVertex3f(0.0f, 1.0f, 0.0f); // Move Up One

// Move Up One Unit

We've placed the first vertex on the screen, setting it's color to red. Now before we place the second vertex we'll change the color to green. That way the second vertex which is the left corner of the triangle will be set to green.

> glColor3f(0.0f,1.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);

// Set The Color To Green // Left And Down Oi Now we're on the third and final vertex. Just before we draw it, we set the color to blue. This will be the right corner of the triangle. As soon as the glEnd() command is issued, the polygon will be filled in. But because it has a different color at each vertex, rather than one solid color throughout, the color will spread out from each corner, eventually meeting in the middle, where the colors will blend together. This is smooth coloring.

```
glColor3f(0.0f,0.0f,1.0f);
glVertex3f( 1.0f,-1.0f, 0.0f);
glEnd();
glTranslatef(3.0f,0.0f,0.0f);
```

// Set The Color To Blue // Right And Down (// Done Drawing A Triangle

```
// From Right Point
```

Now we will draw a solid blue colored square. It's important to remember that anything drawn after the color has been set will be drawn in that color. Every project you create down the road will use coloring in one way or another. Even in scenes where everything is texture mapped, glColor3f can still be used to tint the color of textures, etc. More on that later.

So to draw our square all one color, all we have to do is set the color once to a color we like (blue in this example), then draw the square. The color blue will be used for each vertex because we're not telling OpenGL to change the color at each vertex. The final result... a blue square.

}

Finally change the code to toggle window / fullscreen mode so that the title at the top of the window is proper.

In this tutorial I have tried to explain in as much detail, how to add flat and smooth coloring to your OpenGL polygons. Play around with the code, try changing the red, green and blue values to different numbers. See what colors you can come up with. If you have comments or questions please email me. If you feel I have incorrectly commented something or that the code could be done better in some sections, please let me know. I want to make the best OpenGL tutorials I can. I'm interested in hearing your feedback.

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Peter De Jaegher)
- * DOWNLOAD ASM Code For This Lesson. (Conversion by Foolman)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 4

In the last tutorial I taught you how to add color to triangles and quads. In this tutorial I will teach you how to rotate these colored objects around an axis.

Using the code from the last tutorial, we will be adding to a few places in the code. I will rewrite the entire section of code below so it's easy for you to figure out what's been added, and what needs to be replaced.

We'll start off by adding the two variables to keep track of the rotation for each object. We do this right at the beginning of the program. You'll notice below I've added two lines after BOOL keys[256]. These lines set up two floating point variables that we can use to spin the objects with very fine accuracy. Floating point allows decimal numbers. Meaning we're not stuck using 1, 2, 3 for the angle, we can use 1.1, 1.7, 2.3, or even 1.015 for fine accuracy. You'll find that floating point numbers are essential to OpenGL programming.

#include #include	<windows.h> <gl\gl.h> <gl\glu.h> <gl\glaux.h></gl\glaux.h></gl\glu.h></gl\gl.h></windows.h>	// Header File For // Header File For The Open(// Header File For // Header File For
HDC	hDC=NULL;	// Private GDI Device Contes
HGLRC	hRC=NULL;	// Permanent Rendering Conte
HWND	hWnd=NULL;	// Holds Our Window
bool	keys[256];	// Array Used For 5
bool	active=TRUE;	// Window Active F:
bool	fullscreen=TRUE;	// Fullscreen Flag Set To TH
GLfloat	rtri;	// Angle For The T
GLfloat	rquad;	// Angle For The Q1

Now we need to modify the DrawGLScene() code. I will rewrite the entire procedure. This should make it easier for you to see what changes I have made to the original code. I'll explain why lines have been modified, and what exactly it is that the new lines do. The next section of code is exactly the same as in the last tutorial.

int DrawGLScene(GLvoid)	// Here's Where We
{ glClear(GL_COLOR_BUFFER_BIT GL_DEPTH_BUFFER_BIT);	// Clear The Screen And The
glLoadIdentity();	// Reset The View
glTranslatef(-1.5f,0.0f,-6.0f);	// Move Into The So

The next line of code is new. glRotatef(Angle,Xvector,Yvector,Zvector) is responsible for rotating the object around an axis. You will get alot of use out of this command. Angle is some number (usually stored in a variable) that represents how much you would like to spin the object. Xvector, Yvector and Zvector parameters together represent the vector about which the rotation will occur. If you use values (1,0,0), you are describing a vector which travels in a direction of 1 unit along the x axis towards the right. Values (-1,0,0) describes a vector that travels in a direction of 1 unit along the x axis, but this time towards the left.

D. Michael Traub: has supplied the above explanation of the Xvector, Yvector and Zvector parameters.

To better understand X, Y and Z rotation I'll explain using examples...

X Axis - You're working on a table saw. The bar going through the center of the blade runs left to right (just like the x axis in OpenGL). The sharp teeth spin around the x axis (bar running through the center of the blade), and appear to be cutting towards or away from you depending on which way the blade is being spun. When we spin something on the x axis in OpenGL it will spin the same way.

Y Axis - Imagine that you are standing in the middle of a field. There is a huge tornado coming straight at you. The center of a tornado runs from the sky to the ground (up and down, just like the y axis in OpenGL). The dirt and debris in the tornado spins around the y axis (center of the tornado) from left to right or right to left. When you spin something on the y axis in OpenGL it will spin the same way.

Z Axis - You are looking at the front of a fan. The center of the fan points towards you and away from you (just like the z axis in OpenGL). The blades of the fan spin around the z axis (center of the fan) in a clockwise or counterclockwise direction. When You spin something on the z axis in OpenGL it will spin the same way.

So in the following line of code, if rtri was equal to 7, we would spin 7 on the Y axis (left to right). You can try experimenting with the code. Change the 0.0f's to 1.0f's, and the 1.0f to a 0.0f to spin the triangle on the X and Y axes at the same time.

```
glRotatef(rtri,0.0f,1.0f,0.0f);
```

// Rotate The Tria

The next section of code has not changed. It draws a colorful smooth blended triangle. The triangle will be drawn on the left side of the screen, and will be rotated on it's Y axis causing it to spin left to right.

glBegin(GL_TRIANGLES); glColor3f(1.0f,0.0f,0.0f); glVertex3f(0.0f, 1.0f, 0.0f); glColor3f(0.0f,1.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f); glColor3f(0.0f,0.0f,1.0f); glVertex3f(1.0f,-1.0f, 0.0f); glEnd(); // Start Drawing A
// Set Top Point Of Triangle
 // First Point Of '
// Set Left Point Of Triangl
 // Second Point Of
// Set Right Point Of Triang
 // Third Point Of '
// Done Drawing The Triangle

You'll notice in the code below, that we've added another glLoadldentity(). We do this to reset the view. If we didn't reset the view. If we translated after the object had been rotated, you would get very unexpected results. Because the axis has been rotated, it may not be pointing in the direction you think. So if we translate left on the X axis, we may end up moving up or down instead, depending on how much we've rotated on each axis. Try taking the glLoadldentity() line out to see what I mean.

Once the scene has been reset, so X is running left to right, Y up and down, and Z in and out, we translate. You'll notice we're only moving 1.5 to the right instead of 3.0 like we did in the last lesson. When we reset the screen, our focus moves to the center of the screen. meaning we're no longer 1.5 units to the left, we're back at 0.0. So to get to 1.5 on the right side of zero we dont have to move 1.5 from left to center then 1.5 to the right (total of 3.0) we only have to move from center to the right which is just 1.5 units.

After we have moved to our new location on the right side of the screen, we rotate the quad, on the X axis. This will cause the square to spin up and down.

glLoadIdentity(); glTranslatef(1.5f,0.0f,-6.0f); glRotatef(rquad,1.0f,0.0f,0.0f);

This section of code remains the same. It draws a blue square made from one quad. It will draw the square on the right side of the screen in it's rotated position.

The next two lines are new. Think of rtri, and rquad as containers. At the top of our program we made the containers (GLfloat rtri, and GLfloat rquad). When we built the containers they had nothing in them. The first line below ADDS 0.2 to that container. So each time we check the value in the rtri container after this section of code, it will have gone up by 0.2. The rquad container decreases by 0.15. So every time we check the rquad container, it will have gone down by 0.15. Going down will cause the object to spin the opposite direction it would spin if you were going up.

Try chaning the + to a - in the line below see how the object spins the other direction. Try changing the values from 0.2 to 1.0. The higher the number, the faster the object will spin. The lower the number, the slower it will spin.

rtri+=0.2f; rquad-=0.15f; return TRUE;

}

// Increase The Rot
// Decrease The Rot
// Keep Going

Finally change the code to toggle window / fullscreen mode so that the title at the top of the window is proper.

```
if (keys[VK_F1]) // Is F1 Being Pressed?
{
    keys[VK_F1]=FALSE; // If So Make Key FALSE
    KillGLWindow(); // Kill Our Current
    fullscreen=!fullscreen; // Toggle Fullscree
    // Recreate Our OpenGL Window ( Modified )
    if (!CreateGLWindow("NeHe's Rotation Tutorial",640,480,
    {
        return 0; // Quit If Window Was Not Cn
    }
}
```

In this tutorial I have tried to explain in as much detail as possible, how to rotate objects around an axis. Play around with the code, try spinning the objects, on the Z axis, the X & Y, or all three :) If you have comments or questions please email me. If you feel I have incorrectly commented something or that the code could be done better in some sections, please let me know. I want to make the best OpenGL tutorials I can. I'm interested in hearing your feedback.

Jeff Molofee (NeHe)

* DOWNLOAD Visual C++ Code For This Lesson.

- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Peter De Jaegher)
- * DOWNLOAD ASM Code For This Lesson. (Conversion by Foolman)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 5

Expanding on the last tutorial, we'll now make the object into TRUE 3D object, rather than 2D objects in a 3D world. We will do this by adding a left, back, and right side to the triangle, and a left, right, back, top and bottom to the square. By doing this, we turn the triangle into a pyramid, and the square into a cube.

We'll blend the colors on the pyramid, creating a smoothly colored object, and for the square we'll color each face a different color.

```
int DrawGLScene(GLvoid) // Here's Where We
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The
    glLoadIdentity();
    glTranslatef(-1.5f,0.0f,-6.0f); // Reset The View
    glRotatef(rtri,0.0f,1.0f,0.0f); // Rotate The Pyrat
    glBegin(GL_TRIANGLES); // Start Drawing Tl
```

A few of you have taken the code from the last tutorial, and made 3D objects of your own. One thing I've been asked quite a bit is "how come my objects are not spinning on their axis? It seems like they are spinning all over the screen". In order for your object to spin around an axis, it has to be designed AROUND that axis. You have to remember that the center of any object should be 0 on the X, 0 on the Y, and 0 on the Z.

The following code will create the pyramid around a central axis. The top of the pyramid is one high from the center, the bottom of the pyramid is one down from the center. The top point is right in the middle (zero), and the bottom points are one left from center, and one right from center.

Note that all triangles are drawn in a counterclockwise rotation. This is important, and will be explained in a future tutorial, for now, just know that it's good practice to make objects either clockwise or counterclockwise, but you shouldn't mix the two unless you have a reason to.

We start off by drawing the Front Face. Because all of the faces share the top point, we will make this point red on all of the triangles. The color on the bottom two points of the triangles will alternate. The front face will have a green left point and a blue right point. Then the triangle on the right side will have a blue left point and a green right point. By alternating the bottom two colors on each face, we make a common colored point at the bottom of each face.

 glColor3f(1.0f,0.0f,0.0f);
 // Red

 glVertex3f(0.0f,1.0f,0.0f);
 // Top Of Triangle

 glColor3f(0.0f,1.0f,0.0f);
 // Green

 glVertex3f(-1.0f,-1.0f, 1.0f);
 // Left Of Triangle

 glColor3f(0.0f,0.0f,1.0f);
 // Blue

 glVertex3f(1.0f,-1.0f, 1.0f);
 // Right Of Triangle

Now we draw the right face. Notice then the two bottom point are drawn one to the right of center, and the top point is drawn one up on the y axis, and right in the middle of the x axis. causing the face to slope from center point at the top out to the right side of the screen at the bottom.

Notice the left point is drawn blue this time. By drawing it blue, it will be the same color as the right bottom corner of the front face. Blending blue outwards from that one corner across both the front and right face of the pyramid.

Notice how the remaining three faces are included inside the same glBegin(GL_TRIANGLES) and glEnd() as the first face. Because we're making this entire object out of triangles, OpenGL will know that every three points we plot are the three points of a triangle. Once it's drawn three points, if there are three more points, it will assume another triangle needs to be drawn. If you were to put four points instead of three, OpenGL would draw the first three and assume the fourth point is the start of a new triangle. It would not draw a Quad. So make sure you don't add any extra points by accident.

glColor3f(1.0f,0.0f,0.0f);	11	Red		
glVertex3f(0.0f, 1.0f, 0.0f);			//	Top Of Triangle
glColor3f(0.0f,0.0f,1.0f);	11	Blue		
glVertex3f(1.0f,-1.0f, 1.0f);			//	Left Of Triangle
glColor3f(0.0f,1.0f,0.0f);	11	Green		
glVertex3f(1.0f,-1.0f, -1.0f);			//	Right Of Triang

Now for the back face. Again the colors switch. The left point is now green again, because the corner it shares with the right face is green.

glColor3f(1.0f,0.0f,0.0f);	11	Red		
glVertex3f(0.0f, 1.0f, 0.0f);			//	Top Of Triangle
glColor3f(0.0f,1.0f,0.0f);	//	Green		
glVertex3f(1.0f,-1.0f, -1.0f);			//	Left Of Triangle
glColor3f(0.0f,0.0f,1.0f);	//	Blue		
glVertex3f(-1.0f,-1.0f, -1.0f);			//	Right Of Triang

Finally we draw the left face. The colors switch one last time. The left point is blue, and blends with the right point of the back face. The right point is green, and blends with the left point of the front face.

We're done drawing the pyramid. Because the pyramid only spins on the Y axis, we will never see the bottom, so there is no need to put a bottom on the pyramid. If you feel like experimenting, try adding a bottom using a quad, then rotate on the X axis to see if you've done it correctly. Make sure the color used on each corner of the quad matches up with the colors being used at the four corners of the pyramid.

Now we'll draw the cube. It's made up of six quads. All of the quads are drawn in a counter clockwise order. Meaning the first point is the top right, the second point is the top left, third point is bottom left, and finally bottom right. When we draw the back face, it may seem as though we are drawing clockwise, but you have to keep in mind that if we were behind the cube looking at the front of it, the left side of the screen is actually the right side of the quad, and the right side of the screen would actually be the left side of the quad.

Notice we move the cube a little further into the screen in this lesson. By doing this, the size of the cube appears closer to the size of the pyramid. If you were to move it only 6 units into the screen, the cube would appear much larger than the pyramid, and parts of it might get cut off by the sides of the screen. You can play around with this setting, and see how moving the cube further into the screen makes it appear smaller, and moving it closer makes it appear larger. The reason this happens is perspective. Objects in the distance should appear smaller :)

<pre>glLoadIdentity(); glTranslatef(1.5f,0.0f,-7.0f);</pre>	// Move Right And :
<pre>glRotatef(rquad,1.0f,1.0f,1.0f);</pre>	// Rotate The Cube On X, Y ξ
glBegin(GL_QUADS);	// Start Drawing The Cube

We'll start off by drawing the top of the cube. We move up one unit from the center of the cube. Notice that the Y axis is always one. We then draw a quad on the Z plane. Meaning into the screen. We start off by drawing the top right point of the top of the cube. The top right point would be one unit right, and one unit into the screen. The second point would be one unit to the left, and unit into the screen. Now we have to draw the bottom of the quad towards the viewer. so to do this, instead of going into the screen, we move one unit towards the screen. Make sense?

> glColor3f(0.0f,1.0f,0.0f); // Set glVertex3f(1.0f, 1.0f,-1.0f); glVertex3f(-1.0f, 1.0f,-1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f);

// Set The Color To Blue
 // Top Right Of The
 // Top Left Of The
 // Bottom Left Of '.
 // Bottom Right Of

The bottom is drawn the exact same way as the top, but because it's the bottom, it's drawn down one unit from the center of the cube. Notice the Y axis is always minus one. If we were under the cube, looking at the quad that makes the bottom, you would notice the top right corner is the corner closest to the viewer, so instead of drawing in the distance first, we draw closest to the viewer first, then on the left side closest to the viewer, and then we go into the screen to draw the bottom two points.

If you didn't really care about the order the polygons were drawn in (clockwise or not), you could just copy the same code for the top quad, move it down on the Y axis to -1, and it would work, but ignoring the order the quad is drawn in can cause weird results once you get into fancy things such as texture mapping.

glColor3f(1.0f,0.5f,0.0f); glVertex3f(1.0f,-1.0f, 1.0f); glVertex3f(-1.0f,-1.0f, 1.0f); glVertex3f(-1.0f,-1.0f,-1.0f); glVertex3f(1.0f,-1.0f,-1.0f);

// Set The Color To Orange
 // Top Right Of The
 // Top Left Of The
 // Bottom Left Of `
 // Bottom Right Of

Now we draw the front of the Quad. We move one unit towards the screen, and away from the center to draw the front face. Notice the Z axis is always one. In the pyramid the Z axis was not always one. At the top, the Z axis was zero. If you tried changing the Z axis to zero in the following code, you'd notice that the corner you changed it on would slope into the screen. That's not something we want to do right now :)

 glColor3f(1.0f,0.0f,0.0f);
 // Set The Color To Red

 glVertex3f(1.0f,1.0f,1.0f);
 // Top Right Of The

 glVertex3f(-1.0f,1.0f,1.0f);
 // Top Left Of The

 glVertex3f(1.0f,-1.0f,1.0f);
 // Bottom Left Of The

 glVertex3f(1.0f,-1.0f,1.0f);
 // Bottom Right Of

The back face is a quad the same as the front face, but it's set deeper into the screen. Notice the Z axis is now minus one for all of the points.

glColor3f(1.0f,1.0f,0.0f); // Set The Color To Yellow glVertex3f(1.0f,-1.0f,-1.0f); // Top Right Of The glVertex3f(-1.0f,-1.0f,-1.0f); // Top Left Of The glVertex3f(-1.0f, 1.0f,-1.0f); // Bottom Left Of ' glVertex3f(1.0f, 1.0f,-1.0f); // Bottom Right Of

Now we only have two more quads to draw and we're done. As usual, you'll notice one axis is always the same for all the points. In this case the X axis is always minus one. That's because we're always drawing to the left of center because this is the left face.

glColor3f(0.0f,0.0f,1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); glVertex3f(-1.0f, 1.0f,-1.0f); glVertex3f(-1.0f,-1.0f,-1.0f); glVertex3f(-1.0f,-1.0f, 1.0f);

```
// Set The Color To Blue
    // Top Right Of The
    // Top Left Of The
    // Bottom Left Of '.
    // Bottom Right Of
```

This is the last face to complete the cube. The X axis is always one. Drawing is counter clockwise. If you wanted to, you could leave this face out, and make a box :)

Or if you felt like experimenting, you could always try changing the color of each point on the cube to make it blend the same way the pyramid blends. You can see an example of a blended cube by downloading Evil's first GL demo from my web page. Run it and press TAB. You'll see a beautifully colored cube, with colors flowing across all the faces.

glColor3f(1.0f,0.0f,1.0f); glVertex3f(1.0f, 1.0f,-1.0f); glVertex3f(1.0f, 1.0f, 1.0f); glVertex3f(1.0f,-1.0f, 1.0f); glVertex3f(1.0f,-1.0f,-1.0f); glEnd(); rtri+=0.2f; rquad==0.15f; return TRUE;

```
// Set The Color To Violet
    // Top Right Of The
    // Top Left Of The
    // Bottom Left Of '.
    // Bottom Right Of
// Done Drawing The Quad
    // Increase The Rot
    // Decrease The Rot
    // Keep Going
```

}

By the end of this tutorial, you should have a better understanding of how objects are created in 3D space. You have to think of the OpenGL screen as a giant piece of graph paper, with many transparent layers behind it. Almost like a giant cube made of of points. Some of the points move left to right, some move up and down, and some move further back in the cube. If you can visualize the depth into the screen, you shouldn't have any problems designing new 3D objects.

If you're having a hard time understanding 3D space, don't get frustrated. It can be difficult to grasp right off the start. An object like the cube is a good example to learn from. If you notice, the back face is drawn exactly the same as the front face, it's just further into the screen. Play around with the code, and if you just can't grasp it, email me, and I'll try to answer your questions.

Jeff Molofee (NeHe)

* DOWNLOAD Visual C++ Code For This Lesson.

- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Peter De Jaegher)
- * DOWNLOAD ASM Code For This Lesson. (Conversion by Foolman)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)

* DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)

- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)

* DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 6

Learning how to texture map has many benefits. Lets say you wanted a missile to fly across the screen. Up until this tutorial we'd probably make the entire missile out of polygons, and fancy colors. With texture mapping, you can take a real picture of a missile and make the picture fly across the screen. Which do you think will look better? A photograph or an object made up of triangles and squares? By using texture mapping, not only will it look better, but your program will run faster. The texture mapped missile would only be one quad moving across the screen. A missile made out of polygons could be made up of hundreds or thousands of polygons. The single texture mapped quad will use alot less processing power.

Lets start off by adding five new lines of code to the top of lesson one. The first new line is #include <stdio.h>. Adding this header file allows us to work with files. In order to use fopen() later in the code we need to include this line. Then we add three new floating point variables... xrot, yrot and zrot. These variables will be used to rotate the cube on the x axis, the y axis, and the z axis. The last line GLuint texture[1] sets aside storage space for one texture. If you wanted to load in more than one texture, you would change the number one to the number of textures you wish to load.

<pre>#include #include #include</pre>	<pre><windows.h> <stdio.h> <stdio.h> <sgl\gl.h> <sgl\glu.h> <sgl\glaux.h></sgl\glaux.h></sgl\glu.h></sgl\gl.h></stdio.h></stdio.h></windows.h></pre>	// Header // Header
HDC HGLRC HWND HINSTANC	hDC=NULL; hRC=NULL; hWnd=NULL; EhInstance;	// Privat // Perman // Holds '
bool bool bool	keys[256]; active=TRUE; fullscreen=TRUE;	// Fullsc:
GLfloat GLfloat GLfloat	<pre>xrot; yrot; zrot;</pre>	
GLuint	<pre>texture[1];</pre>	
LRESULT	CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);	// Declara

Now immediately after the above code, and before ReSizeGLScene(), we want to add the following section of code. The job of this code is to load in a bitmap file. If the file doesn't exist NULL is sent back meaning the texture couldn't be loaded. Before I start explaining the code there are a few VERY important things you need to know about the images you plan to use as textures. The image height and width MUST be a power of 2. The width and height must be at least 64 pixels, and for compatability reasons, shouldn't be more than 256 pixels. If the image you want to use is not 64, 128 or 256 pixels on the width or height, resize it in an art program. There are ways around this limitation, but for now we'll just stick to standard texture sizes.

First thing we do is create a file handle. A handle is a value used to identify a resource so that our

program can access it. We set the handle to NULL to start off.

AUX_RGBImageRec *LoadBMP(char *Filename)	// Loads .
<pre> FILE *File=NULL; </pre>	// File Ha

Next we check to make sure that a filename was actually given. The person may have use LoadBMP() without specifying the file to load, so we have to check for this. We don't want to try loading nothing :)

```
if (!Filename)
{
    return NULL;
}
```

If a filename was given, we check to see if the file exists. The line below tries to open the file.

```
File=fopen(Filename,"r");
```

If we were able to open the file it obviously exists. We close the file with fclose(File) then we return the image data. auxDIBImageLoad(Filename) reads in the data.

```
if (File) // Does Tl
{
    fclose(File);
    return auxDIBImageLoad(Filename); // Load Tl
}
```

If we were unable to open the file we'll return NULL. which means the file couldn't be loaded. Later on in the program we'll check to see if the file was loaded. If it wasn't we'll quit the program with an error message.

return NULL;

}

This is the section of code that loads the bitmap (calling the code above) and converts it into a texture.

int LoadGLTextures()
{

// Check '

We'll set up a variable called **Status**. We'll use this variable to keep track of whether or not we were able to load the bitmap and build a texture. We set Status to FALSE (meaning nothing has been loaded or built) by default.

int Status=FALSE;

Now we create an image record that we can store our bitmap in. The record will hold the bitmap width, height, and data.

AUX_RGBImageRec *TextureImage[1];

We clear the image record just to make sure it's empty.

```
memset(TextureImage,0,sizeof(void *)*1);
```

Now we load the bitmap and convert it to a texture. TextureImage[0]=LoadBMP("Data/NeHe.bmp") will jump to our LoadBMP() code. The file named NeHe.bmp in the Data directory will be loaded. If everything goes well, the image data is stored in TextureImage[0], **Status** is set to TRUE, and we start to build our texture.

```
// Load The Bitmap, Check For Errors, If Bitmap's Not Found Quit
if (TextureImage[0]=LoadBMP("Data/NeHe.bmp"))
{
    Status=TRUE;
```

Now that we've loaded the image data into TextureImage[0], we will build a texture using this data. The first line glGenTextures(1, &texture[0]) tells OpenGL we want to build one texture (increase the number if you load more than one texture), and we want the texture to be stored in slot 0 of **texture** []. Remember at the very beginning of this tutorial we created room for one texture with the line GLuint **texture[1]**. Although you'd think the first texture would be stored at &texture[1] instead of &texture[0], it wont work. The first actual storage area is 0. If we wanted two textures we would use GLuint texture[2] and the second texture would be stored at texture[1].

The second line glBindTexture(GL_TEXTURE_2D, texture[0]) tells OpenGL that texture[0] (the first texture) will be a 2D texture. 2D textures have both height (on the Y axes) and width (on the X axes). The main function of glBindTexture is to point OpenGL to available memory. In this case we're telling OpenGL there is memory available at &texture[0]. When we create the texture, it will be stored in this memory space. Basically glBindTexture() points to ram that holds or will hold our texture.

glGenTextures(1, &texture[0]);
// Typical Texture Generation Using Data From The Bitmap
glBindTexture(GL_TEXTURE_2D, texture[0]);

// Status

// Create

// Set The

Next we create the actual texture. The following line tells OpenGL the texture will be a 2D texture (GL_TEXTURE_2D). Zero represents the images level of detail, this is usually left at zero. Three is the number of data components. Because the image is made up of red data, green data and blue data, there are three components. TextureImage[0]->sizeX is the width of the texture. If you know the width, you can put it here, but it's easier to let the computer figure it out for you. TextureImage [0]->sizey is the height of the texture. zero is the border. It's usually left at zero. GL_RGB tells OpenGL the image data we are using is made up of red, green and blue data in that order. GL_UNSIGNED_BYTE means the data that makes up the image is made up of unsigned bytes, and finally... TextureImage[0]->data tells OpenGL where to get the texture data from. In this case it points to the data stored in the TextureImage[0] record.

// Generate The Texture
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]

The next two lines tell OpenGL what type of filtering to use when the image is larger (GL_TEXTURE_MAG_FILTER) or stretched on the screen than the original texture, or when it's smaller (GL_TEXTURE_MIN_FILTER) on the screen than the actual texture. I usually use GL_LINEAR for both. This makes the texture look smooth way in the distance, and when it's up close to the screen. Using GL_LINEAR requires alot of work from the processor/video card, so if your system is slow, you might want to use GL_NEAREST. A texture that's filtered with GL_NEAREST will appear blocky when it's stretched. You can also try a combination of both. Make it filter things up close, but not things in the distance.

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR); // Linear
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR); // Linear
```

Now we free up any ram that we may have used to store the bitmap data. We check to see if the bitmap data was stored in TextureImage[0]. If it was we check to see if the data has been stored. If data was stored, we erase it. Then we free the image structure making sure any used memory is freed up.

Finally we return the status. If everything went OK, the variable **Status** will be TRUE. If anything went wrong, **Status** will be FALSE.

return Status;

}

}

// If Tex

I've added a few lines of code to InitGL. I'll repost the entire section of code, so it's easy to see the lines that I've added, and where they go in the code. The first line if (!LoadGLTextures()) jumps to the routine above which loads the bitmap and makes a texture from it. If LoadGLTextures() fails for any reason, the next line of code will return FALSE. If everything went OK, and the texture was created, we enable 2D texture mapping. If you forget to enable texture mapping your object will usually appear solid white, which is definitely not good.

```
int InitGL(GLvoid)
                                                                                    // All Se
{
         if (!LoadGLTextures())
         {
                  return FALSE;
         }
                                                                                    // Enable
         glEnable(GL_TEXTURE_2D);
         glShadeModel(GL_SMOOTH);
                                                                                    // Enable
         glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
         glClearDepth(1.0f);
         glEnable(GL_DEPTH_TEST);
                                                                                    // Enable
         glDepthFunc(GL_LEQUAL);
         glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
                                                                                    // Really
         return TRUE;
```

}

Now we draw the textured cube. You can replace the DrawGLScene code with the code below, or you can add the new code to the original lesson one code. This section will be heavily commented so it's easy to understand. The first two lines of code glClear() and glLoadIdentity() are in the original lesson one code. glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) will clear the screen to the color we selected in InitGL(). In this case, the screen will be cleared to blue. The depth buffer will also be cleared. The view will then be reset with glLoadIdentity().

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear;
    glLoadIdentity(); // Reset '
    glTranslatef(0.0f,0.0f,-5.0f);
```

The following three lines of code will rotate the cube on the x axis, then the y axis, and finally the z axis. How much it rotates on each axis will depend on the value stored in xrot, yrot and zrot.

```
glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);
glRotatef(zrot,0.0f,0.0f,1.0f);
```

The next line of code selects which texture we want to use. If there was more than one texture you wanted to use in your scene, you would select the texture using glBindTexture(GL_TEXTURE_2D, texture[*number of texture to use*]). If you wanted to change textures, you would bind to the new texture. One thing to note is that you can NOT bind a texture inside glBegin() and glEnd(), you have to do it before or after glBegin(). Notice how we use glBindTextures to specify which texture to create and to select a specific texture.

glBindTexture(GL_TEXTURE_2D, texture[0]);

To properly map a texture onto a quad, you have to make sure the top right of the texture is mapped to the top right of the quad. The top left of the texture is mapped to the top left of the quad, the bottom right of the texture is mapped to the bottom left of the quad, and finally, the bottom left of the texture is mapped to the bottom left of the quad. If the corners of the texture do not match the same corners of the quad, the image may appear upside down, sideways, or not at all.

The first value of glTexCoord2f is the X coordinate. 0.0f is the left side of the texture. 0.5f is the middle of the texture, and 1.0f is the right side of the texture. The second value of glTexCoord2f is the Y coordinate. 0.0f is the bottom of the texture. 0.5f is the middle of the texture, and 1.0f is the top of the texture.

So now we know the top left coordinate of a texture is 0.0f on X and 1.0f on Y, and the top left vertex of a quad is -1.0f on X, and 1.0f on Y. Now all you have to do is match the other three texture coordinates up with the remaining three corners of the quad.

Try playing around with the x and y values of glTexCoord2f. Changing 1.0f to 0.5f will only draw the left half of a texture from 0.0f (left) to 0.5f (middle of the texture). Changing 0.0f to 0.5f will only draw the right half of a texture from 0.5f (middle) to 1.0f (right).

glBegin(GL_QUADS); // Front Face glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f); // Bottom glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f); // Top Ri glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Le: // Back Face glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Rig glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f); // Top Le: glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, -1.0f); // Bottom // Top Face glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Le: glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Bottom glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, 1.0f, 1.0f); // Bottom glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f); // Top Rig // Bottom Face glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Top Rig glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, -1.0f, -1.0f); // Top Le: glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f); // Bottom glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom // Right face glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, -1.0f, -1.0f); // Bottom glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f); // Top Rig glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f); // Top Le: glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f); // Bottom // Left Face glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Rig glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Le:

Now we increase the value of xrot, yrot and zrot. Try changing the number each variable increases by to make the cube spin faster or slower, or try changing a + to a - to make the cube spin the other direction.

```
xrot+=0.3f;
yrot+=0.2f;
zrot+=0.4f;
return true;
```

}

You should now have a better understanding of texture mapping. You should be able to texture map the surface of any quad with an image of your choice. Once you feel confident with your understanding of 2D texture mapping, try adding six different textures to the cube.

Texture mapping isn't to difficult to understand once you understand texture coordinates. If you're having problems understanding any part of this tutorial, let me know. Either I'll rewrite that section of the tutorial, or I'll reply back to you in email. Have fun creating texture mapped scenes of your own :)

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Brad Choate)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 7

In this tutorial I'll teach you how to use three different texture filters. I'll teach you how to move an object using keys on the keyboard, and I'll also teach you how to apply simple lighting to your OpenGL scene. Lots covered in this tutorial, so if the previous tutorials are giving you problems, go back and review. It's important to have a good understanding of the basics before you jump into the following code.

We're going to be modifying the code from lesson one again. As usual, if there are any major changes, I will write out the entire section of code that has been modified. We'll start off by adding a few new variables to the program.

<pre>#include #include #include</pre>	<pre>e <windows.h> e <stdio.h> e <gl\gl.h> e <gl\glu.h> e <gl\glu.h> e <gl\glaux.h></gl\glaux.h></gl\glu.h></gl\glu.h></gl\gl.h></stdio.h></windows.h></pre>	// Header // Header
HDC HGLRC HWND HINSTANC	hDC=NULL; hRC=NULL; hWnd=NULL; EhInstance;	// Privat // Perman // Holds '
bool bool bool	keys[256]; active=TRUE; fullscreen=TRUE;	// Fullsc:

The lines below are new. We're going to add three boolean variables. BOOL means the variable can only be TRUE or FALSE. We create a variable called **light** to keep track of whether or not the lighting is on or off. The variables **Ip** and **fp** are used to store whether or not the 'L' or 'F' key has been pressed. I'll explain why we need these variables later on in the code. For now, just know that they are important.

BOOL	light;
BOOL	lp;
BOOL	fp;

Now we're going to set up five variables that will control the angle on the x axis (**xrot**), the angle on the y axis (**yrot**), the speed the crate is spinning at on the x axis (**xspeed**), and the speed the crate is spinning at on the y axis (**yspeed**). We'll also create a variable called **z** that will control how deep into the screen (on the z axis) the crate is.

GLfloat xrot; GLfloat yrot; GLfloat xspeed; GLfloat yspeed;

GLfloat z=-5.0f;

Now we set up the arrays that will be used to create the lighting. We'll use two different types of

light. The first type of light is called ambient light. Ambient light is light that doesn't come from any particular direction. All the objects in your scene will be lit up by the ambient light. The second type of light is called diffuse light. Diffuse light is created by your light source and is reflected off the surface of an object in your scene. Any surface of an object that the light hits directly will be very bright, and areas the light barely gets to will be darker. This creates a nice shading effect on the sides of our crate.

Light is created the same way color is created. If the first number is 1.0f, and the next two are 0.0f, we will end up with a bright red light. If the third number is 1.0f, and the first two are 0.0f, we will have a bright blue light. The last number is an alpha value. We'll leave it at 1.0f for now.

So in the line below, we are storing the values for a white ambient light at half intensity (0.5f). Because all the numbers are 0.5f, we will end up with a light that's halfway between off (black) and full brightness (white). Red, blue and green mixed at the same value will create a shade from black (0.0f) to white(1.0f). Without an ambient light, spots where there is no diffuse light will appear very dark.

GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f };

In the next line we're storing the values for a super bright, full intensity diffuse light. All the values are 1.0f. This means the light is as bright as we can get it. A diffuse light this bright lights up the front of the crate nicely.

GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f };

Finally we store the position of the light. The first three numbers are the same as glTranslate's three numbers. The first number is for moving left and right on the x plane, the second number is for moving up and down on the y plane, and the third number is for moving into and out of the screen on the z plane. Because we want our light hitting directly on the front of the crate, we don't move left or right so the first value is 0.0f (no movement on x), we don't want to move up and down, so the second value is 0.0f as well. For the third value we want to make sure the light is always in front of the crate. So we'll position the light off the screen, towards the viewer. Lets say the glass on your monitor is at 0.0f on the z plane. We'll position the light at 2.0f on the z plane. If you could actually see the light, it would be floating in front of the glass on your monitor. By doing this, the only way the light would be behind the crate is if the crate was also in front of the glass on your monitor. Of course if the crate was no longer behind the glass on your monitor, you would no longer see the crate, so it doesn't matter where the light is. Does that make sense?

There's no real easy way to explain the third parameter. You should know that -2.0f is going to be closer to you than -5.0f. and -100.0f would be WAY into the screen. Once you get to 0.0f, the image is so big, it fills the entire monitor. Once you start going into positive values, the image no longer appears on the screen cause it has "gone past the screen". That's what I mean when I say out of the screen. The object is still there, you just can't see it anymore.

Leave the last number at 1.0f. This tells OpenGL the designated coordinates are the position of the light source. More about this in a later tutorial.

// Ambien

// Depth

The **filter** variable below is to keep track of which texture to display. The first texture (texture 0) is made using gl_nearest (no smoothing). The second texture (texture 1) uses gl_linear filtering which smooths the image out quite a bit. The third texture (texture 2) uses mipmapped textures, creating a very nice looking texture. The variable **filter** will equal 0, 1 or 2 depending on the texture we want to use. We start off with the first texture.

GLuint texture[3] creates storage space for the three different textures. The textures will be stored at texture[0], texture[1] and texture[2].

GLuint filter; GLuint texture[3];

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// Declara

Now we load in a bitmap, and create three different textures from it. This tutorial uses the glaux library to load in the bitmap, so make sure you have the glaux library included before you try compiling the code. I know Delphi, and Visual C++ both have glaux libraries. I'm not sure about other languages. I'm only going to explain what the new lines of code do, if you see a line I haven't commented on, and you're wondering what it does, check tutorial six. It explains loading, and building texture maps from bitmap images in great detail.

Immediately after the above code, and before ReSizeGLScene(), we want to add the following section of code. This is the same code we used in lesson 6 to load in a bitmap file. Nothing has changed. If you're not sure what any of the following lines do, read tutorial six. It explains the code below in detail.

```
AUX_RGBImageRec *LoadBMP(char *Filename)
                                                                                       // Loads 1
{
         FILE *File=NULL;
                                                                                       // File Ha
         if (!Filename)
         {
                   return NULL;
         }
         File=fopen(Filename, "r");
                                                                                       // Check '
         if (File)
                                                                                       // Does Tl
         {
                   fclose(File);
                                                                                       // Load Tl
                   return auxDIBImageLoad(Filename);
         }
         return NULL;
```

```
}
```

This is the section of code that loads the bitmap (calling the code above) and converts it into 3 textures. Status is used to keep track of whether or not the texture was loaded and created.

```
int LoadGLTextures()
{
    int Status=FALSE;
```

// Status

```
AUX_RGBImageRec *TextureImage[1];// Creatememset(TextureImage,0,sizeof(void *)*1);// Set Th
```

Now we load the bitmap and convert it to a texture. TextureImage[0]=LoadBMP("Data/Crate.bmp") will jump to our LoadBMP() code. The file named Crate.bmp in the Data directory will be loaded. If everything goes well, the image data is stored in TextureImage[0], **Status** is set to TRUE, and we start to build our texture.

```
// Load The Bitmap, Check For Errors, If Bitmap's Not Found Quit
if (TextureImage[0]=LoadBMP("Data/Crate.bmp"))
{
    Status=TRUE;
```

Now that we've loaded the image data into TextureImage[0], we'll use the data to build 3 textures. The line below tells OpenGL we want to build three textures, and we want the texture to be stored in texture[0], texture[1] and texture[2].

```
glGenTextures(3, &texture[0]);
```

In tutorial six, we used linear filtered texture maps. They require a hefty amount of processing power, but they look real nice. The first type of texture we're going to create in this tutorial uses GL_NEAREST. Basically this type of texture has no filtering at all. It takes very little processing power, and it looks real bad. If you've ever played a game where the textures look all blocky, it's probably using this type of texture. The only benefit of this type of texture is that projects made using this type of texture will usually run pretty good on slow computers.

You'll notice we're using GL_NEAREST for both the MIN and MAG. You can mix GL_NEAREST with GL_LINEAR, and the texture will look a bit better, but we're intested in speed, so we'll use low quality for both. The MIN_FILTER is the filter used when an image is drawn smaller than the original texture size. The MAG_FILTER is used when the image is bigger than the original texture size.

```
// Create Nearest Filtered Texture
glBindTexture(GL_TEXTURE_2D, texture[0]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST); ( NEW )
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST); ( NEW )
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]
```

The next texture we build is the same type of texture we used in tutorial six. Linear filtered. The only thing that has changed is that we are storing this texture in texture[1] instead of texture[0] because it's our second texture. If we stored it in texture[0] like above, it would overwrite the GL_NEAREST texture (the first texture).

// Create Linear Filtered Texture
glBindTexture(GL_TEXTURE_2D, texture[1]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]

}

Now for a new way to make textures. Mipmapping! You may have noticed that when you make an image very tiny on the screen, alot of the fine details disappear. Patterns that used to look nice start looking real bad. When you tell OpenGL to build a mipmapped texture OpenGL tries to build different sized high quality textures. When you draw a mipmapped texture to the screen OpenGL will select the BEST looking texture from the ones it built (texture with the most detail) and draw it to the screen instead of resizing the original image (which causes detail loss).

I had said in tutorial six there was a way around the 64,128,256,etc limit that OpenGL puts on texture width and height. gluBuild2DMipmaps is it. From what I've found, you can use any bitmap image you want (any width and height) when building mipmapped textures. OpenGL will automatically size it to the proper width and height.

Because this is texture number three, we're going to store this texture in texture[2]. So now we have texture[0] which has no filtering, texture[1] which uses linear filtering, and texture[2] which uses mipmapped textures. We're done building the textures for this tutorial.

// Create MipMapped Texture
glBindTexture(GL_TEXTURE_2D, texture[2]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAR);

The following line builds the mipmapped texture. We're creating a 2D texture using three colors (red, green, blue). TextureImage[0]->sizeX is the bitmaps width, TextureImage[0]->sizeY is the bitmaps height, GL_RGB means we're using Red, Green, Blue colors in that order. GL_UNSIGNED_BYTE means the data that makes the texture is made up of bytes, and TextureImage[0]->data points to the bitmap data that we're building the texture from.

```
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[
```

Now we free up any ram that we may have used to store the bitmap data. We check to see if the bitmap data was stored in TextureImage[0]. If it was we check to see if the data has been stored. If data was stored, we erase it. Then we free the image structure making sure any used memory is freed up.

// If Tex

Finally we return the status. If everything went OK, the variable **Status** will be TRUE. If anything went wrong, **Status** will be FALSE.

return Status;

}

Now we load the textures, and initialize the OpenGL settings. The first line of InitGL loads the textures using the code above. After the textures have been created, we enable 2D texture mapping with glEnable(GL_TEXTURE_2D). The shade mode is set to smooth shading, The background color is set to black, we enable depth testing, then we enable nice perspective calculations.

```
int InitGL(GLvoid)
                                                                                    // All Se
         if (!LoadGLTextures())
         {
                  return FALSE;
         }
         glEnable(GL_TEXTURE_2D);
                                                                                    // Enable
         glShadeModel(GL_SMOOTH);
                                                                                    // Enable
         glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
         glClearDepth(1.0f);
         glEnable(GL_DEPTH_TEST);
                                                                                    // Enable
         glDepthFunc(GL_LEQUAL);
         glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
                                                                                    // Really
```

Now we set up the lighting. The line below will set the amount of ambient light that light1 will give off. At the beginning of this tutorial we stored the amount of ambient light in LightAmbient. The values we stored in the array will be used (half intensity ambient light).

glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);

Next we set up the amount of diffuse light that light number one will give off. We stored the amount of diffuse light in LightDiffuse. The values we stored in this array will be used (full intensity white light).

glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);

Now we set the position of the light. We stored the position in LightPosition. The values we stored in this array will be used (right in the center of the front face, 0.0f on x, 0.0f on y, and 2 unit towards the viewer {coming out of the screen} on the z plane).

glLightfv(GL_LIGHT1, GL_POSITION,LightPosition);

// Positi

Finally, we enable light number one. We haven't enabled GL_LIGHTING though, so you wont see any lighting just yet. The light is set up, and positioned, it's even enabled, but until we enable GL_LIGHTING, the light will not work.

```
glEnable(GL_LIGHT1);
return TRUE;
```

}

In the next section of code, we're going to draw the texture mapped cube. I will comment a few of the line only because they are new. If you're not sure what the uncommented lines do, check tutorial number six.

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear '
    glLoadIdentity(); // Reset '
```

The next three lines of code position and rotate the texture mapped cube. glTranslatef(0.0f,0.0f,z) moves the cube to the value of z on the z plane (away from and towards the viewer). glRotatef (xrot,1.0f,0.0f,0.0f) uses the variable **xrot** to rotate the cube on the x axis. glRotatef (yrot,1.0f,0.0f,0.0f) uses the variable **yrot** to rotate the cube on the y axis.

glTranslatef(0.0f,0.0f,z);

glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);

The next line is similar to the line we used in tutorial six, but instead of binding texture[0], we are binding texture[filter]. Any time we press the 'F' key, the value in filter will increase. If this value is higher than two, the variable filter is set back to zero. When the program starts the filter will be set to zero. This is the same as saying glBindTexture(GL_TEXTURE_2D, texture[0]). If we press 'F' once more, the variable filter will equal one, which is the same as saying glBindTexture (GL_TEXTURE_2D, texture[1]). By using the variable filter we can select any of the three textures we've made.

glBindTexture(GL_TEXTURE_2D, texture[filter]);

glBegin(GL_QUADS);

glNormal3f is new to my tutorials. A normal is a line pointing straight out of the middle of a polygon at a 90 degree angle. When you use lighting, you need to specify a normal. The normal tells OpenGL which direction the polygon is facing... which way is up. If you don't specify normals, all kinds of weird things happen. Faces that shouldn't light up will light up, the wrong side of a polygon will light up, etc. The normal should point outwards from the polygon.

Looking at the front face you'll notice that the normal is positive on the z axis. This means the normal is pointing at the viewer. Exactly the direction we want it pointing. On the back face, the normal is pointing away from the viewer, into the screen. Again exactly what we want. If the cube is spun 180 degrees on either the x or y axis, the front will be facing into the screen and the back will be facing towards the viewer. No matter what face is facing the viewer, the normal of that face will also be pointing towards the viewer. Because the light is close to the viewer, any time the normal is pointing towards the viewer it's also pointing towards the light. When it does, the face will light up. The more a normal points towards the light, the brighter that face is. If you move into the center of the cube you'll notice it's dark. The normals are point out, not in, so there's no light inside the box,

// Transla

// Start 1

exactly as it should be.

// Front Face glNormal3f(0.0f, 0.0f, 1.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f); // Point glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f); // Point glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point // Back Face glNormal3f(0.0f, 0.0f, -1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f); // Point // Point glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, -1.0f); // Top Face glNormal3f(0.0f, 1.0f, 0.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point // Point glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, 1.0f, 1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f); // Point // Bottom Face glNormal3f(0.0f,-1.0f, 0.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, -1.0f, -1.0f); // Point glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f); // Point glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point // Right face glNormal3f(1.0f, 0.0f, 0.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f(1.0f, -1.0f, -1.0f); // Point glTexCoord2f(1.0f, 1.0f); glVertex3f(1.0f, 1.0f, -1.0f); // Point glTexCoord2f(0.0f, 1.0f); glVertex3f(1.0f, 1.0f, 1.0f); // Point // Point glTexCoord2f(0.0f, 0.0f); glVertex3f(1.0f, -1.0f, 1.0f); // Left Face glNormal3f(-1.0f, 0.0f, 0.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point // Point glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Done D: glEnd();

J____() :

The next two lines increase **xrot** and **yrot** by the amount stored in **xspeed**, and **yspeed**. If the value in **xspeed** or **yspeed** is high, **xrot** and **yrot** will increase quickly. The faster **xrot**, or **yrot** increases, the faster the cube spins on that axis.

xrot+=xspeed; yrot+=yspeed; return TRUE;

}

Now we move down to WinMain(). Were going to add code to turn lighting on and off, spin the crate, change the filter and move the crate into and out of the screen. Closer to the bottom of WinMain() you will see the command SwapBuffers(hDC). Immediately after this line, add the following code.

This code checks to see if the letter 'L' has been pressed on the keyboard. The first line checks to see if 'L' is being pressed. If 'L' is being pressed, but **Ip** isn't false, meaning 'L' has already been pressed once or it's being held down, nothing will happen.

SwapBuffers(hDC);
if (keys['L'] && !lp)
{

If **Ip** was false, meaning the 'L' key hasn't been pressed yet, or it's been released, **Ip** becomes true. This forces the person to let go of the 'L' key before this code will run again. If we didn't check to see if the key was being held down, the lighting would flicker off and on over and over, because the program would think you were pressing the 'L' key over and over again each time it came to this section of code.

-

Once **Ip** has been set to true, telling the computer that 'L' is being held down, we toggle lighting off and on. The variable **light** can only be true of false. So if we say **light=!light**, what we are actually saying is light equals NOT light. Which in english translates to if **light** equals true make **light** not true (false), and if **light** equals false, make **light** not false (true). So if **light** was true, it becomes false, and if **light** was false it becomes true.

> lp=TRUE; light=!light;

// lp Bec

Now we check to see what **light** ended up being. The first line translated to english means: If **light** equals false. So if you put it all together, the lines do the following: If **light** equals false, disable lighting. This turns all lighting off. The command 'else' translates to: if it wasn't false. So if **light** wasn't false, it must have been true, so we turn lighting on.

```
if (!light)
{
     glDisable(GL_LIGHTING);
}
else
{
     glEnable(GL_LIGHTING);
}
```

The following line checks to see if we stopped pressing the 'L' key. If we did, it makes the variable **Ip** equal false, meaning the 'L' key isn't pressed. If we didn't check to see if the key was released, we'd be able to turn lighting on once, but because the computer would always think 'L' was being held down so it wouldn't let us turn it back off.

}

// Swap B

Now we do something similar with the 'F' key. if the key is being pressed, and it's not being held down or it's never been pressed before, it will make the variable **fp** equal true meaning the key is now being held down. It will then increase the variable called **filter**. If **filter** is greater than 2 (which would be texture[3], and that texture doesn't exist), we reset the variable **filter** back to zero.

```
if (keys['F'] && !fp)
{
         fp=TRUE;
                                               // fp Bec
         filter+=1;
         if (filter>2)
         {
                  filter=0;
                                               // If So,
         }
}
if (!keys['F'])
{
         fp=FALSE;
                                               // If So,
}
```

The next four lines check to see if we are pressing the 'Page Up' key. If we are it decreases the variable z. If this variable decreases, the cube will move into the distance because of the glTranslatef(0.0f,0.0f,z) command used in the DrawGLScene procedure.

```
if (keys[VK_PRIOR])
{
    z-=0.02f; // If So,
}
```

These four lines check to see if we are pressing the 'Page Down' key. If we are it increases the variable z and moves the cube towards the viewer because of the glTranslatef(0.0f,0.0f,z) command used in the DrawGLScene procedure.

```
if (keys[VK_NEXT]) // Is Pag'
{
        z+=0.02f; // If So,
}
```

Now all we have to check for is the arrow keys. By pressing left or right, **xspeed** is increased or decreased. By pressing up or down, **yspeed** is increased or decreased. Remember further up in the tutorial I said that if the value in **xspeed** or **yspeed** was high, the cube would spin faster. The longer you hold down an arrow key, the faster the cube will spin in that direction.

```
}
if (keys[VK_RIGHT])
{
    yspeed+=0.01f;
}
if (keys[VK_LEFT]) // Is Lef
{
    yspeed-=0.01f;
}
```

Like all the previous tutorials, make sure the title at the top of the window is correct.

```
if (keys[VK_F1])
                                                                            // Is F1 1
                            {
                                      keys[VK_F1]=FALSE;
                                                                            // If So l
                                      KillGLWindow();
                                      fullscreen=!fullscreen;
                                      // Recreate Our OpenGL Window
                                      if (!CreateGLWindow("NeHe's Textures, Lighting
                                      {
                                               return 0;
                                                                            // Quit I:
                                      }
                            }
                   }
         }
}
// Shutdown
KillGLWindow();
return (msg.wParam);
```

```
}
```

By the end of this tutorial you should be able to create and interact with high quality, realistic looking, textured mapped objects made up of quads. You should understand the benefits of each of the three filters used in this tutorial. By pressing specific keys on the keyboard you should be able to interact with the object(s) on the screen, and finally, you should know how to apply simple lighting to a scene making the scene appear more realistic.

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Brad Choate)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 8

Simple Transparency

Most special effects in OpenGL rely on some type of blending. Blending is used to combine the color of a given pixel that is about to be drawn with the pixel that is already on the screen. How the colors are combined is based on the alpha value of the colors, and/or the blending function that is being used. Alpha is a 4th color component usually specified at the end. In the past you have used GL_RGB to specify color with 3 components. GL_RGBA can be used to specify alpha as well. In addition, we can use glColor4f() instead of glColor3f().

Most people think of Alpha as how opaque a material is. An alpha value of 0.0 would mean that the material is completely transparent. A value of 1.0 would be totally opaque.

The Blending Equation

If you are uncomfortable with math, and just want to see how to do transparency, skip this section. If you want to understand how blending works, this section is for you.

(Rs Sr + Rd Dr, Gs Sg + Gd Dg, Bs Sb + Bd Db, As Sa + Ad Da)

OpenGL will calculate the result of blending two pixels based on the above equation. The s and r subscripts specify the source and destination pixels. The S and D components are the blend factors. These values indicate how you would like to blend the pixels. The most common values for S and D are (As, As, As, As) (AKA source alpha) for S and (1, 1, 1, 1) - (As, As, As, As) (AKA one minus src alpha) for D. This will yield a blending equation that looks like this:

(Rs As + Rd (1 - As), Gs As + Gd (1 - As), Bs As + Bs (1 - As), As As + Ad (1 - As))

This equation will yield transparent/translucent style effects.

Blending in OpenGL

We enable blending just like everything else. Then we set the equation, and turn off depth buffer writing when drawing transparent objects, since we still want objects behind the translucent shapes to be drawn. This isn't the proper way to blend, but most the time in simple projects it will work fine.

Rui Martins Adds: The correct way is to draw all the transparent (with alpha < 1.0) polys after you have drawn the entire scene, and to draw them in reverse depth order (farthest first).

This is due to the fact that blending two polygons (1 and 2) in different order gives different results, i.e. (assuming poly 1 is nearest to the viewer, the correct way would be to draw poly 2 first and then poly 1. If you look at it, like in reality, all the light comming from behind these two polys (which are transparent) has to pass poly 2 first and then poly 1 before it reaches the eye of the viewer.

You should SORT THE TRANSPARENT POLYGONS BY DEPTH and draw them AFTER THE ENTIRE SCENE HAS BEEN DRAWN, with the DEPTH BUFFER ENABLED, or you will get incorrect results. I know this sometimes is a pain, but this is the correct way to do it.

We'll be using the code from lesson seven. We start off by adding two new variables to the top of the code. I'll rewrite the entire section of code for clarity.

```
#include <windows.h>
                                                              // Header File For Windows
                                                     // Header File For Standard Input/Out
#include <stdio.h>
                                                     // Header File For The OpenGL32 Libra
#include <gl\gl.h>
                                                              // Header File For The GLu32
#include <gl\glu.h>
#include <gl\glaux.h>
                                                              // Header File For The GLaux
                hDC=NULL;
HDC
                                                     // Private GDI Device Context
       یلانی hRC=NULL;
-
HGLRC
                                                     // Permanent Rendering Context
HWND
                hWnd=NULL;
                                                              // Holds Our Window Handle
HINSTANCEhInstance;
                                                     // Holds The Instance Of The Applicat
bool keys[256];
                                                              // Array Used For The Keyboa
bool
       active=TRUE;
                                                              // Window Active Flag Set To
bool fullscreen=TRUE;
bool light;
                                                     // Fullscreen Flag Set To Fullscreen
                                                              // Lighting ON/OFF
bool blend;
                                                              // Blending OFF/ON? ( NEW )
bool lp;
                                                              // L Pressed?
bool
       fp;
                                                              // F Pressed?
bool bp;
                                                              // B Pressed? ( NEW )
GLfloat xrot;
                                                              // X Rotation
GLfloat yrot;
                                                              // Y Rotation
GLfloat xspeed;
                                                              // X Rotation Speed
GLfloat yspeed;
                                                              // Y Rotation Speed
GLfloat z=-5.0f;
                                                     // Depth Into The Screen
GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f }; // Ambient Light Values
GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f }; // Diffuse Light Values
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f }; // Light Position
GLuint filter;
                                                              // Which Filter To Use
GLuint texture[3];
                                                              // Storage for 3 textures
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc
```

Move down to LoadGLTextures(). Find the line that says texture1 = auxDIBImageLoad ("Data/crate.bmp"), change it to the line below. We're using a stained glass type texture for this tutorial instead of the crate texture.

texturel = auxDIBImageLoad("Data/glass.bmp"); // Load The Glass Bitmap (MODIFIED)

Add the following two lines somewhere in the InitGL() section of code. What this line does is sets the drawing brightness of the object to full brightness with 50% alpha (opacity). This means when blending is enabled, the object will be 50% transparent. The second line sets the type of blending we're going to use.

Rui Martins Adds: An alpha value of 0.0 would mean that the material is completely transparent. A value of 1.0 would be totally opaque.

glColor4f(1.0f,1.0f,1.0f,0.5f); // Full Brightness, 50% Alp}
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Blending Function For Translucency

Look for the following section of code, it can be found at the very bottom of lesson seven.

```
if (keys[VK_LEFT]) // Is Left Arrow Being Pressed?
{
    yspeed-=0.01f; // If So, Decrease yspeed
}
```

Right under the above code, we want to add the following lines. The lines below watch to see if the 'B' key has been pressed. If it has been pressed, the computer checks to see if blending is off or on. If blending is on, the computer turns it off. If blending was off, the computer will turn it on.

```
if (keys['B'] && !bp)
                                                      // Is B Key Pressed And bp H
{
        bp=TRUE;
                                            // If So, bp Becomes TRUE
        blend = !blend;
                                                     // Toggle blend TRUE / FALSE
                                            // Is blend TRUE?
        if(blend)
         {
                                                     // Turn Blending On
                 glEnable(GL_BLEND);
                 glDisable(GL_DEPTH_TEST); // Turn Depth Testing Off
        }
        else
                                                      // Otherwise
         {
                 glDisable(GL_BLEND);
                                                     // Turn Blending Off
                 glEnable(GL_DEPTH_TEST); // Turn Depth Testing On
        }
}
if (!keys['B'])
                                                      // Has B Key Been Released?
{
        bp=FALSE;
                                           // If So, bp Becomes FALSE
}
```

But how can we specify the color if we are using a texture map? Simple, in modulated texture mode, each pixel that is texture mapped is multiplied by the current color. So, if the color to be drawn is (0.5, 0.6, 0.4), we multiply it times the color and we get (0.5, 0.6, 0.4, 0.2) (alpha is assumed to be 1.0 if not specified).

Thats it! Blending is actually quite simple to do in OpenGL.

Note (11/13/99)

I (NeHe) have modified the blending code so the output of the object looks more like it should. Using Alpha values for the source and destination to do the blending will cause artifacting. Causing back faces to appear darker, along with side faces. Basically the object will look very screwy. The way I do blending may not be the best way, but it works, and the object appears to look like it should when lighting is enabled. Thanks to Tom for the initial code, the way he was blending was the proper way to blend with alpha values, but didn't look as attractive as people expected :)

The code was modified once again to address problems that some video cards had with glDepthMask(). It seems this command would not effectively enable and disable depth buffer testing on some cards, so I've changed back to the old fashioned glEnable and Disable of Depth Testing.

Alpha from texture map.

The alpha value that is used for transparency can be read from a texture map just like color, to do this, you will need to get alpha into the image you want to load, and then use GL_RGBA for the color format in calls to glTexImage2D().

Questions?

If you have any questions, feel free to contact me at stanis@cs.wisc.edu.

Tom Stanis

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 9

Welcome to Tutorial 9. By now you should have a very good understanding of OpenGL. You've learned everything from setting up an OpenGL Window, to texture mapping a spinning object while using lighting and blending. This will be the first semi-advanced tutorial. You'll learn the following: Moving bitmaps around the screen in 3D, removing the black pixels around the bitmap (using blending), adding color to a black & white texture and finally you'll learn how to create fancy colors and simple animation by mixing different colored textures together.

We'll be modifying the code from lesson one for this tutorial. We'll start off by adding a few new variables to the beginning of the program. I'll rewrite the entire section of code so it's easier to see where the changes are being made.

#include <windows.h>
#include <stdio.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>

HDC hDC=NULL; HGLRC hRC=NULL; HWND hWnd=NULL; HINSTANCEhInstance;

bool keys[256]; bool active=TRUE; bool fullscreen=TRUE;

// Window Active F: // Fullscreen Flag Set To Fu

The following lines are new. **twinkle** and **tp** are BOOLean variables meaning they can be TRUE or FALSE. **twinkle** will keep track of whether or not the twinkle effect has been enabled. **tp** is used to check if the 'T' key has been pressed or released. (pressed **tp**=TRUE, relased **tp**=FALSE).

BOOL twinkle; BOOL tp; // Twinkling Stars // 'T' Key Pressed'

num will keep track of how many stars we draw to the screen. It's defined as a CONSTant. This means it can never change within the code. The reason we define it as a constant is because you can not redefine an array. So if we've set up an array of only 50 stars and we decided to increase **num** to 51 somewhere in the code, the array can not grow to 51, so an error would occur. You can change this value to whatever you want it to be in this line only. Don't try to change the value of **num** later on in the code unless you want disaster to occur.

const num=50;

// Number Of Stars

Now we create a structure. The word structure sounds intimidating, but it's not really. A structure is a group simple data (variables, etc) representing a larger similar group. In english :) We know that we're keeping track of stars. You'll see that the 7th line below is **stars**;. We know each star will have 3 values for color, and all these values will be integer values. The 3rd line int **r**,**g**,**b** sets up 3 integer values. One for red (**r**), one for green (**g**), and one for blue (**b**). We know each star will be a different distance from the center of the screen, and can be place at one of 360 different angles from the center. If you look at the 4th line below, we make a floating point value called **dist**. This will keep track of the distance. The 5th line creates a floating point value called **angle**. This will keep track of the stars angle.

So now we have this group of data that describes the color, distance and angle of a star on the screen. Unfortunately we have more than one star to keep track of. Instead of creating 50 red values, 50 green values, 50 blue values, 50 distance values and 50 angle values, we just create an array called **star**. Each number in the **star** array will hold all of the information in our structure called **stars**. We make the **star** array in the 8th line below. If we break down the 8th line: **stars star[num]**. This is what we come up with. The type of array is going to be **stars**. **stars** is a structure. So the array is going to hold all of the information in the structure. The name of the array is **star**. The number of arrays is **[num]**. So because **num**=50, we now have an array called **star**. Our array stores the elements of the structure **stars**. Alot easier than keeping track of each star with seperate variables. Which would be a very stupid thing to do, and would not allow us to add remove stars by changing the const value of **num**.

// Create A Structu
// Stars Color
// Stars Distance I
// Stars Current Au
// Structures Name
// Make 'star' Array Of 'nur

Next we set up variables to keep track of how far away from the stars the viewer is (**zoom**), and what angle we're seeing the stars from (**tilt**). We make a variable called **spin** that will spin the twinkling stars on the z axis, which makes them look like they are spinning at their current location.

loop is a variable we'll use in the program to draw all 50 stars, and **texture[1]** will be used to store the one b&w texture that we load in. If you wanted more textures, you'd increase the value from one to however many textures you decide to use.

 GLfloat zoom=-15.0f;
 // Viewing Distance

 GLfloat tilt=90.0f;
 // Tilt The View

 GLfloat spin;
 // Spin Twinkling !

 GLuint loop;
 // General Loop Vai

 GLuint texture[1];
 // Storage For One

 LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
 // Declaration For WndProc

Right after the line above we add code to load in our texture. I shouldn't have to explain the code in great detail. It's the same code we used to load the textures in lesson 6, 7 and 8. The bitmap we load this time is called star.bmp. We generate only one texture using glGenTextures(1, &texture [0]). The texture will use linear filtering.

```
AUX_RGBImageRec *LoadBMP(char *Filename)
                                                                // Loads A Bitmap Image
{
         FILE *File=NULL;
                                                                // File Handle
         if (!Filename)
                                                                         // Make Sure A File
         {
                  return NULL;
                                                                         // If Not Return NI
         }
         File=fopen(Filename, "r");
                                                                // Check To See If The File
         if (File)
                                                                // Does The File Exist?
         {
                  fclose(File);
                                                                         // Close The Handle
                  return auxDIBImageLoad(Filename);
                                                              // Load The Bitmap And Retur
         }
         return NULL;
                                                                         // If Load Failed I
```

}

This is the section of code that loads the bitmap (calling the code above) and converts it into a textures. **Status** is used to keep track of whether or not the texture was loaded and created.

```
int LoadGLTextures()
                                                                         // Load Bitmaps And
ł
         int Status=FALSE;
                                                                // Status Indicator
        AUX_RGBImageRec *TextureImage[1];
                                                                // Create Storage Space For
        memset(TextureImage,0,sizeof(void *)*1);
                                                                // Set The Pointer To NULL
         // Load The Bitmap, Check For Errors, If Bitmap's Not Found Quit
         if (TextureImage[0]=LoadBMP("Data/Star.bmp"))
                  Status=TRUE;
                                                                         // Set The Status 1
                  glGenTextures(1, &texture[0]);
                                                                         // Create One Text
                  // Create Linear Filtered Texture
                  glBindTexture(GL_TEXTURE_2D, texture[0]);
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
                  glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]
         }
         if (TextureImage[0])
                                                                         // If Texture Exist
         {
                  if (TextureImage[0]->data)
                                                               // If Texture Image Exists
                  {
                           free(TextureImage[0]->data);
                                                                         // Free The Texture
                  }
                  free(TextureImage[0]);
                                                                         // Free The Image :
```

```
Jeff Molofee's OpenGL Windows Tutorial #9
```

}

```
}
return Status; // Return The Statu
```

Now we set up OpenGL to render the way we want. We're not going to be using Depth Testing in this project, so make sure if you're using the code from lesson one that you remove glDepthFunc (GL_LEQUAL); and glEnable(GL_DEPTH_TEST); otherwise you'll see some very bad results. We're using texture mapping in this code however so you'll want to make sure you add any lines that are not in lesson 1. You'll notice we're enabling texture mapping, along with blending.

int InitGL(GLvoid)	// All Setup For OpenGL Goes
if (!LoadGLTextures()) {	// Jump To Texture
return FALSE; }	// If Texture Didn
<pre>glEnable(GL_TEXTURE_2D); glShadeModel(GL_SMOOTH); glClearColor(0.0f, 0.0f, 0.0f, 0.5f); glClearDepth(1.0f); glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); glBlerdEvmg(CL_SEG_ALDUA_CL_ONE);</pre>	<pre>// Enable Texture Mapping // Enable Smooth Shading</pre>
glBlendFunc(GL_SRC_ALPHA,GL_ONE); glEnable(GL_BLEND);	// Set The Blending Functior // Enable Blending

The following code is new. It sets up the starting angle, distance, and color of each star. Notice how easy it is to change the information in the structure. The loop will go through all 50 stars. To change the angle of **star[1]** all we have to do is say **star[1].angle**={some number}. It's that simple!

<pre>for (loop=0; loop<num; loop++)="" pre="" {<=""></num;></pre>	// Create A Loop Tl
<pre>star[loop].angle=0.0f;</pre>	// Start All The St

I calculate the distance by taking the current star (which is the value of **loop**) and dividing it by the maximum amount of stars there can be. Then I multiply the result by 5.0f. Basically what this does is moves each star a little bit farther than the previous star. When **loop** is 50 (the last star), **loop** divided by **num** will be 1.0f. The reason I multiply by 5.0f is because 1.0f*5.0f is 5.0f. 5.0f is the very edge of the screen. I don't want stars going off the screen so 5.0f is perfect. If you set the **zoom** further into the screen you could use a higher number than 5.0f, but your stars would be alot smaller (because of perspective).

You'll notice that the colors for each star are made up of random values from 0 to 255. You might be wondering how we can use such large values when normally the colors are from 0.0f to 1.0f. When we set the color we'll use glColor4ub instead of glColor4f. ub means Unsigned Byte. A byte can be any value from 0 to 255. In this program it's easier to use bytes than to come up with a random floating point value.

```
star[loop].dist=(float(loop)/num)*5.0f;// Calculate Distanstar[loop].r=rand()%256;// Give star[loop] A Randomstar[loop].g=rand()%256;// Give star[loop] A Randomstar[loop].b=rand()%256;// Give star[loop] A Random
```

return TRUE;

}

The Resize code is the same, so we'll jump to the drawing code. If you're using the code from lesson one, delete the DrawGLScene code, and just copy what I have below. There's only 2 lines of code in lesson one anyways, so there's not a lot to delete.

```
int DrawGLScene(GLvoid) // Here's Where We
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The
    glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Our Texture
    for (loop=0; loop<num; loop++) // Loop Through Al:
        glLoadIdentity(); // Reset The View Before We
        glTranslatef(0.0f,0.0f,zoom); // Zoom Into The Sc
        glRotatef(tilt,1.0f,0.0f,0.0f); // Tilt The View (I</pre>
```

Now we move the star. The star starts off in the middle of the screen. The first thing we do is spin the scene on the y axis. If we spin 90 degrees, the x axis will no longer run left to right, it will run into and out of the screen. As an example to help clarify. Imagine you were in the center of a room. Now imagine that the left wall had -x written on it, the front wall had -z written on it, the right wall had +x written on it, and the wall behind you had +z written on it. If the room spun 90 degrees to the right, but you did not move, the wall in front of you would no longer say -z it would say -x. All of the walls would have moved. -z would be in front +z behind, -x would be in front, and +x would be behind you. Make sense? By rotating the scene, we change the direction of the x and z planes.

The next line of code moves to a positive value on the x plane. Normally a positive value on x would move us to the right side of the screen (where +x usually is), but because we've rotated on the y plane, the +x could be anywhere. If we rotated by 180 degrees, it would be on the left side of the screen instead of the right. So when we move forward on the positive x plane, we could be moving left, right, forward or backward.

```
glRotatef(star[loop].angle,0.0f,1.0f,0.0f); // Rotate To The Current Sta
glTranslatef(star[loop].dist,0.0f,0.0f); // Move Forward On The X Pla
```

Now for some tricky code. The star is actually a flat texture. Now if you drew a flat quad in the middle of the screen and texture mapped it, it would look fine. It would be facing you like it should. But if you rotated on the y axis by 90 degrees, the texture would be facing the right and left sides of the screen. All you'd see is a thin line. We don't want that to happen. We want the stars to face the screen all the time, no matter how much we rotate and tilt the screen.

We do this by cancelling any rotations that we've made, just before we draw the star. You cancel the rotations in reverse order. So above we tilted the screen, then we rotated to the stars current angle. In reverse order, we'd un-rotate (new word) the stars current angle. To do this we use the negative value of the angle, and rotate by that. So if we rotated the star by 10 degrees, rotating it back -10 degrees will make the star face the screen once again on that axis. So the first line below cancels the rotation on the y axis. Now we need to until the screen on the x axis. So to do that we just tilt the screen by -tilt. After we've cancelled the x and y rotations, the star will face the screen completely.

glRotatef(-star[loop].angle,0.0f,1.0f,0.0f); // Cancel The Current Stars
glRotatef(-tilt,1.0f,0.0f,0.0f); // Cancel The Screen Tilt

If twinkle is TRUE, we'll draw a non-spinning star on the screen. To get a different color, we take the maximum number of stars (num) and subtract the current stars number (loop), then subtract 1 because our loop only goes from 0 to num-1. If the result was 10 we'd use the color from star number 10. That way the color of the two stars is usually different. Not a good way to do it, but effective. The last value is the alpha value. The lower the value, the darker the star is.

If twinkle is enabled, each star will be drawn twice. This will slow down the program a little depending on what type of computer you have. If twinkle is enabled, the colors from the two stars will mix together creating some really nice colors. Also because this star does not spin, it will appear as if the stars are animated when twinkling is enabled. (look for yourself if you don't understand what I mean).

Notice how easy it is to add color to the texture. Even though the texture is black and white, it will become whatever color we select before we draw the texture. Also take note that we're using bytes for the color values rather than floating point numbers. Even the alpha value is a byte.

```
if (twinkle)
                                                       // Twinkling Stars
{
         // Assign A Color Using Bytes
         glColor4ub(star[(num-loop)-1].r,star[(num-loop)-1].g,star[(num-l
         glBegin(GL_QUADS);
                                             // Begin Drawing The Texture
                  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 0.0f)
                  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 0.0f)
                  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f)
                  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f)
         qlEnd();
                                              // Done Drawing The Texture
}
```

Now we draw the main star. The only difference from the code above is that this star is always drawn, and this star spins on the z axis.

```
glRotatef(spin,0.0f,0.0f,1.0f);
                                                       // Rotate The Star
// Assign A Color Using Bytes
glColor4ub(star[loop].r,star[loop].g,star[loop].b,255);
glBegin(GL_QUADS);
                                              // Begin Drawing The Texture
         glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
         glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 0.0f);
         glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
         glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd();
                                              // Done Drawing The Textured
```

Here's where we do all the movement. We spin the normal stars by increasing the value of **spin**. Then we change the angle of each star. The angle of each star is increased by **loop/num**. What this does is spins the stars that are farther from the center faster. The stars closer to the center spin slower. Finally we decrease the distance each star is from the center of the screen. This makes the stars look as if they are being sucked into the middle of the screen.

```
spin+=0.01f;
star[loop].angle+=float(loop)/num; // Changes The Angle Of A St
star[loop].dist-=0.01f;
```

// Used To Spin The // Changes The Dist The lines below check to see if the stars have hit the center of the screen or not. When a star hits the center of the screen it's given a new color, and is moved 5 units from the center, so it can start it's journey back to the center as a new star.

```
if (star[loop].dist<0.0f) // Is The Star In The Middle
{
    star[loop].dist+=5.0f; // Move The Star 5
    star[loop].r=rand()%256; // Give It A New Red Value
    star[loop].g=rand()%256; // Give It A New Green Value
    star[loop].b=rand()%256; // Give It A New Blue Value
  }
}
return TRUE; // Everything Went</pre>
```

```
}
```

Now we're going to add code to check if any keys are being pressed. Go down to WinMain(). Look for the line SwapBuffers(hDC). We'll add our key checking code right under that line. lines of code.

The lines below check to see if the T key has been pressed. If it has been pressed and it's not being held down the following will happen. If **twinkle** is FALSE, it will become TRUE. If it was TRUE, it will become FALSE. Once T is pressed **tp** will become TRUE. This prevents the code from running over and over again if you hold down the T key.

```
SwapBuffers(hDC); // Swap Buffers (Double Bufi
if (keys['T'] && !tp) // Is T Being Press
{
    tp=TRUE; // If So, Make tp TRUE
    twinkle=!twinkle; // Make twinkle Equal The OF
}
```

The code below checks to see if you've let go of the T key. If you have, it makes **tp**=FALSE. Pressing the T key will do nothing unless **tp** is FALSE, so this section of code is very important.

if (!keys['T'])	// Has The T Key Be
{ tp=FALSE; }	// If So, make tp FALSE

The rest of the code checks to see if the up arrow, down arrow, page up or page down keys are being pressed.

```
if (keys[VK_UP]) // Is Up Arrow Being Pressed
{
    tilt-=0.5f; // Tilt The Screen
}
if (keys[VK_DOWN]) // Is Down Arrow Being Press
{
    tilt+=0.5f; // Tilt The Screen
```

```
}
if (keys[VK_PRIOR]) // Is Page Up Being
{
    zoom-=0.2f; // Zoom Out
}
if (keys[VK_NEXT]) // Is Page Down Being Presse
{
    zoom+=0.2f; // Zoom In
}
```

Like all the previous tutorials, make sure the title at the top of the window is correct.

```
if (keys[VK_F1])
                                                     // Is F1 Being Pressed?
         {
                 keys[VK_F1]=FALSE;
                                                   // If So Make Key FALSE
                 KillGLWindow();
                                                            // Kill Our Current
                 fullscreen=!fullscreen;
                                                              // Toggle Fullscre
                 // Recreate Our OpenGL Window
                 if (!CreateGLWindow("NeHe's Textures, Lighting & Keyboard Tutori
                 {
                          return 0;
                                                    // Quit If Window Was Not Ci
                 }
        }
}
```

}

In this tutorial I have tried to explain in as much detail how to load in a gray scale bitmap image, remove the black space around the image (using blending), add color to the image, and move the image around the screen in 3D. I've also shown you how to create beautiful colors and animation by overlapping a second copy of the bitmap on top of the original bitmap. Once you have a good understanding of everything I've taught you up till now, you should have no problems making 3D demos ofyour own. All the basics have been covered!

Jeff Molofee (NeHe)

* DOWNLOAD Visual C++ Code For This Lesson.

- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Solaris Code For This Lesson. (Conversion by Lakmal Gunasekara)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD BeOS Code For This Lesson. (Conversion by Chris Herborth)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 10

This tutorial was created by Lionel Brits (Betelgeuse). This lesson only explains the sections of code that have been added. By adding just the lines below, the program will not run. If you're interested to know where each of the lines of code below go, download the source code, and follow through it, as you read the tutorial.

Welcome to the infamous Tutorial 10. By now you have a spinning cube or a couple of stars, and you have the basic *feel* for 3D programming. But wait! Don't run off and start to code Quake IV just yet. Spinning cubes just aren't going to make cool deathmatch opponents :-) These days you need a large, complicated and dynamic 3D world with 6 degrees of freedom and fancy effects like mirrors, portals, warping and of course, high framerates. This tutorial explains the basic "structure" of a 3D world, and also how to move around in it.

Data structure

While it is perfectly alright to code a 3D environment as a long series of numbers, it becomes increasingly hard as the complexity of the environment goes up. For this reason, we must catagorize our data into a more workable fashion. At the top of our list is the sector. Each 3D world is basically a collection of sectors. A sector can be a room, a cube, or any enclosed volume.

typedef struct tagSECTOR	// Build Our Sector
{	
int numtriangles;	// Number Of Triang
TRIANGLE* triangle;	// Pointe:
} SECTOR;	// Call It SECTOR

A sector holds a series of polygons, so the next catagory will be the triangle (we will stick to triangles for now, as they are alot easier to code.)

typedef struct tagTRIANGLE	// Build Our Triang
<pre>VERTEX vertex[3]; } TRIANGLE;</pre>	// Array Of Three // Call I

The triangle is basically a polygon made up of vertices (plural of vertex), which brings us to our last catagory. The vertex holds the real data that OpenGL is interested in. We define each point on the triangle with it's position in 3D space (x, y, z) as well as it's texture coordinates (u, v).

```
typedef struct tagVERTEX
{
    float x, y, z;
    float u, v;
} VERTEX;
```

Loading files

Storing our world data inside our program makes our program quite static and boring. Loading worlds from disk, however, gives us much more flexibility as we can test different worlds without having to recompile our program. Another advantage is that the user can interchange worlds and modify them without having to know the in's and out's of our program. The type of data file we are going to be using will be text. This makes for easy editing, and less code. We will leave binary files for a later date.

The question is, how do we get our data from our file. First, we create a new function called *SetupWorld*(). We define our file as *filein*, and we open it for read-only access. We must also close our file when we are done. Let us take a look at the code so far:

```
// Previous Declaration: char* worldfile = "data\\world.txt";
void SetupWorld() // Setup Our World
{
    FILE *filein; // File T
    filein = fopen(worldfile, "rt"); // Open Our File
    ...
    (read our data)
    ...
    fclose(filein); // Close (
    return; // Jump B;
}
```

Our next challenge is to read each individual line of text into a variable. This can be done in a number of ways. One problem is that not all lines in the file will contain meaningful information. Blank lines and comments shouldn't be read. Let us create a function called *readstr()*. This function will read one meaningful line of text into an initialised string. Here's the code:

Next, we must read in the sector data. This lesson will deal with one sector only, but it is easy to implement a multi-sector engine. Let us turn back to *SetupWorld()*.Our program must know how many triangles are in our sector. In our data file, we will define the number of triangles as follows:

NUMPOLLIES n

Here's the code to read the number of triangles:

Jeff Molofee's OpenGL Windows Tutorial #10 (By Lionel Brits)

```
char oneline[255]; // String To Store
...
readstr(filein,oneline); // Get Single Line
sscanf(oneline, "NUMPOLLIES %d\n", &numtriangles); // Read In Number (
```

The rest of our world-loading process will use the same process. Next, we initialize our sector and read some data into it:

```
// Previous Declaration: SECTOR sector1;
char oneline[255];
                                                                                        // String To Store
int numtriangles;
                                                                                        // Number Of Triang
                                                                                                   // 3D And
float x, y, z, u, v;
. . .
sector1.triangle = new TRIANGLE[numtriangles];
                                                                                                   // Alloca
                                                                                        // Define The Numbe
sector1.numtriangles = triangles;
// Step Through Each Triangle In Sector
for (int triloop = 0; triloop < numtriangles; triloop++)</pre>
                                                                                        // Loop Through Al:
{
           // Step Through Each Vertex In Triangle
           for (int vertloop = 0; vertloop < 3; vertloop++)</pre>
                                                                                        // Loop Through Al.
           {
                     readstr(filein,oneline);
                                                                                        // Read String To I
                      // Read Data Into Respective Vertex Values
                     // Store Values Into Respective Vertices
                     sector1.triangle[triloop].vertex[vertloop].x = x;
                                                                                      // Sector 1, Triang
                     sector1.triangle[triloop].vertex[vertloop].y = y; // Sector 1, Triang
sector1.triangle[triloop].vertex[vertloop].z = z; // Sector 1, Triang
sector1.triangle[triloop].vertex[vertloop].u = u; // Sector 1, Triang
sector1.triangle[triloop].vertex[vertloop].v = v; // Sector 1, Triang
           }
}
```

Each triangle in our data file is declared as follows:

 X1
 X1
 U1
 V1

 X2
 Y2
 Z2
 U2
 V2

 X3
 Y3
 Z3
 U3
 V3

Displaying Worlds

Now that we can load our sector into memory, we need to display it on screen. So far we have done some minor rotations and translations, but our camera was always centered at the origin (0,0,0). Any good 3D engine would have the user be able to walk around and explore the world, and so will ours. One way of doing this is to move the camera around and draw the 3D environment relative to the camera position. This is slow and hard to code. What we will do is this:

- 1. Rotate and translate the camera position according to user commands
- 2. Rotate the world around the origin in the opposite direction of the camera rotation (giving the illusion that the camera has been rotated)
- 3. Translate the world in the opposite manner that the camera has been translated (again, giving the illusion that the camera has moved)

This is pretty simple to implement. Let's start with the first stage (Rotation and translation of the camera).

```
if (keys[VK_RIGHT])
                                                                                     // Is The
{
         yrot -= 1.5f;
                                                                                     // Rotate
}
                                                                           // Is The Left Arro
if (keys[VK_LEFT])
{
         yrot += 1.5f;
                                                                                     // Rotate
}
if (keys[VK_UP])
                                                                           // Is The Up Arrow
ł
         xpos -= (float)sin(yrot*piover180) * 0.05f;
                                                                           // Move On The X-P:
         zpos -= (float)cos(yrot*piover180) * 0.05f;
                                                                           // Move On The Z-P.
         if (walkbiasangle >= 359.0f)
                                                                                    // Is wall
         {
                  walkbiasangle = 0.0f;
                                                                                     // Make w
         }
         else
                                                                                     // Otherw
         {
                   walkbiasangle+= 10;
                                                                                     // If wall
         }
         walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
                                                                                     // Causes
}
if (keys[VK_DOWN])
                                                                           // Is The Down Arro
         xpos += (float)sin(yrot*piover180) * 0.05f;
                                                                           // Move On The X-P.
         zpos += (float)cos(yrot*piover180) * 0.05f;
                                                                           // Move On The Z-P.
         if (walkbiasangle <= 1.0f)
                                                                           // Is walkbiasangle
         {
                  walkbiasangle = 359.0f;
                                                                                     // Make wa
         }
         else
                                                                                     // Otherw
         {
                  walkbiasangle-= 10;
                                                                                     // If wall
         }
         walkbias = (float)sin(walkbiasangle * piover180)/20.0f;
                                                                                     // Causes
}
```

That was fairly simple. When either the left or right cursor key is pressed, the rotation variable *yrot* is incremented or decremented appropriatly. When the forward or backwards cursor key is pressed, a new location for the camera is calculated using the sine and cosine calculations (some trigonometry required :-). *Piover180* is simply a conversion factor for converting between degrees and radians.

Next you ask me: What is this walkbias? It's a word I invented :-) It's basically an offset that occurs when a person walks around (head bobbing up and down like a buoy. It simply adjusts the camera's Y position with a sine wave. I had to put this in, as simply moving forwards and backwards didn't look to great.

Now that we have these variables down, we can proceed with steps two and three. This will be done in the display loop, as our program isn't complicated enough to merit a seperate function.

```
int DrawGLScene(GLvoid) // Draw T
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And
    glLoadIdentity(); // Reset The Curren
```

```
GLfloat x_m, y_m, z_m, u_m, v_m;
                                                                      // Floating Point 1
GLfloat xtrans = -xpos;
                                                                                // Used F
GLfloat ztrans = -zpos;
                                                                                // Used F
GLfloat ytrans = -walkbias-0.25f;
                                                                      // Used For Bounci
                                                                      // 360 Degree Angle
GLfloat sceneroty = 360.0f - yrot;
int numtriangles;
                                                                      // Integer To Hold
glRotatef(lookupdown,1.0f,0,0);
                                                                                // Rotate
glRotatef(sceneroty,0,1.0f,0);
                                                                                // Rotate
glTranslatef(xtrans, ytrans, ztrans);
                                                                                // Transla
                                                                                // Select
glBindTexture(GL_TEXTURE_2D, texture[filter]);
numtriangles = sector1.numtriangles;
                                                                      // Get The Number (
// Process Each Triangle
for (int loop_m = 0; loop_m < numtriangles; loop_m++)</pre>
                                                                      // Loop Through Al:
          glBegin(GL_TRIANGLES);
                                                                                // Start 1
                    glNormal3f( 0.0f, 0.0f, 1.0f);
                                                                                // Normal
                    x_m = sector1.triangle[loop_m].vertex[0].x; // X Vertex Of 1st
                    y_m = sector1.triangle[loop_m].vertex[0].y; // Y Vertex Of 1st
                    z_m = sector1.triangle[loop_m].vertex[0].z; // Z Vertex Of 1st
                    u_m = sector1.triangle[loop_m].vertex[0].u; // U Texture Coord
v_m = sector1.triangle[loop_m].vertex[0].v; // V Texture Coord
                    glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);
                                                                            // Set Th
                    x_m = sector1.triangle[loop_m].vertex[1].x; // X Vertex Of 2nd
                    y_m = sector1.triangle[loop_m].vertex[1].y; // Y Vertex Of 2nd
                    z_m = sector1.triangle[loop_m].vertex[1].z; // Z Vertex Of 2nd
                    u_m = sector1.triangle[loop_m].vertex[1].u; // U Texture Coord
v_m = sector1.triangle[loop_m].vertex[1].v; // V Texture Coord
                    glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);
                                                                            // Set Th
                    x_m = sector1.triangle[loop_m].vertex[2].x; // X Vertex Of 3rd
                    y_m = sector1.triangle[loop_m].vertex[2].y; // Y Vertex Of 3rd
                    z_m = sector1.triangle[loop_m].vertex[2].z; // Z Vertex Of 3rd
                    u_m = sector1.triangle[loop_m].vertex[2].u; // U Texture Coord
v_m = sector1.triangle[loop_m].vertex[2].v; // V Texture Coord
                    glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // Set Th
                                                                      // Done Drawing Tr:
          glEnd();
return TRUE;
                                                                                // Jump Bi
```

```
}
```

And voila! We have drawn our first frame. This isn't exactly Quake but hey, we aren't exactly Carmack's or Abrash's. While running the program, you may want to press F, B, PgUp and PgDown to see added effects. PgUp/Down simply tilts the camera up and down (the same process as panning from side to side.) The texture included is simply a mud texture with a bumpmap of my school ID picture; that is, if NeHe decided to keep it :-).

So now you're probably thinking where to go next. Don't even consider using this code to make a full-blown 3D engine, since that's not what it's designed for. You'll probably want more than one sector in your game, especially if you're going to implement portals. You'll also want to have polygons with more than 3 vertices, again, essential for portal engines. My current implementation of this code allows for multiple sector loading and does backface culling (not drawing polygons that face away from the camera). I'll write a tutorial on that soon, but as it uses alot of math, I'm going to write a tutorial on matrices first.

NeHe (05/01/00): I've added FULL comments to each of the lines listed in this tutorial. Hopefully things make more sense now. Only a few of the lines had comments after them, now they all do :)

Please, if you have any problems with the code/tutorial (this is my first tutorial, so my explanations are a little vague), don't hesitate to email me mailto:iam@cadvision.com Until next time,

Lionel Brits (ßetelgeuse)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 11

Well greetings all. For those of you that want to see what we are doing here, you can check it out at the end of my demo/hack Worthless! I am bosco and I will do my best to teach you guys how to do the animated, sine-wave picture. This tutorial is based on NeHe's tutorial #6 and you should have at least that much knowledge. You should download the source package and place the bitmap I've included in a directory called data where your source code is. Or use your own texture if it's an appropriate size to be used as a texture with OpenGL.

First things first. Open Tutorial #6 in Visual C++ and add the following include statement right after the other #include statements. The #include below allows us to work with complex math such as sine and cosine.

#include <math.h>

// For The Sin() Function

We'll use the array **points** to store the individual x, y & z coordinates of our grid. The grid is 45 points by 45 points, which in turn makes 44 guads x 44 guads. wiggle count will be used to keep track of how fast the texture waves. Every three frames looks pretty good, and the variable hold will store a floating point value to smooth out the waving of the flag. These lines can be added at the top of the program, somewhere under the last #include line, and before the GLuint texture[1] line.

float points[45][45][3]; int wiggle_count = 0; GLfloat hold;

// The Array For Tl // Counter Used To // Temporarily Hold

Move down the the LoadGLTextures() procedure. We want to use the texture called Tim.bmp. Find LoadBMP("Data/NeHe.bmp") and replace it with LoadBMP("Data/Tim.bmp").

if (TextureImage[0]=LoadBMP("Data/Tim.bmp")) // Load The Bitmap

Now add the following code to the bottom of the InitGL() function before return TRUE.

// Back Face Is Filled In glPolygonMode(GL_BACK, GL_FILL); glPolygonMode(GL_FRONT, GL_LINE); // Front Face Is Drawn With These simply specify that we want back facing polygons to be filled completely and that we want front facing polygons to be outlined only. Mostly personal preference at this point. Has to do with the orientation of the polygon or the direction of the vertices. See the Red Book for more information on this. Incidentally, while I'm at it, let me plug the book by saying it's one of the driving forces behind me learning OpenGL, not to mention NeHe's site! Thanks NeHe. Buy The Programmer's Guide to OpenGL from Addison-Wesley. It's an invaluable resource as far as I'm concerned. Ok, back to the tutorial.

Right below the code above, and above return TRUE, add the following lines.

```
// Loop Through The X Plane
for(float float_x = 0.0f; float_x < 9.0f; float_x += 0.2f)
{
    // Loop Through The Y Plane
    for( float float_y = 0.0f; float_y < 9.0f; float_y += 0.2f)
    {
        // Apply The Wave To Our Mesh
        points[ int(float_x*5.0f) ][ int(float_y*5.0f) ][0] = float_x -
        points[ int(float_x*5.0f) ][ int(float_y*5.0f) ][1] = float_y -
        points[ int(float_x*5.0f) ][ int(float_y*5.0f) ][2] = float(sin(
     }
}</pre>
```

Ok, before I said our grid is 45 points by 45 points. Well to accomplish this without having to push our scene back too far, we merely use a world coordinate of 9x9 and space the points 0.2 units apart.

The two loops above initialize the points on our grid. I initialize variables in my loop to localize them in my mind as merely loop variables. Not sure it's kosher. To come up with the array reference we have to multiply our loop variable by 5 (i.e. 45 / 9 = 5). I subtract 4.4 from each of the coordinates to put the "wave" centered on the origin. The same effect could be accomplished with a translate, but I prefer this method.

The final value points[x][y][2] statement is our sine value. The sin() function requires radians. We take our degree value, which is our float_x multiplied by 40.0f. Once we have that, to convert to radians we take the degree, divide by 360.0f, multiply by pi, or an approximation and then multiply by 2.0f.

I'm going to re-write the DrawGLScene function from scratch so clean it out and it replace with the following code.

```
int DrawGLScene(GLvoid) // Draw Our GL Scen
{
    int x, y;
    float float_x, float_y, float_xb, float_yb;
    // Loop Variables
    // Used To Break The Flag In
```

Different variables used for controlling the loops. See the code below but most of these serve no "specific" purpose other than controlling loops and storing temporary values.

glClear(GL_COLOR_BUFFER_BIT	GL_DEPTH_BUFFER_BIT);	// Clear The Screen And Dept
glLoadIdentity();		// Reset The Current Matrix

```
glTranslatef(0.0f,0.0f,-12.0f); // Translate 17 Un:
glRotatef(xrot,1.0f,0.0f,0.0f); // Rotate On The X
glRotatef(yrot,0.0f,1.0f,0.0f); // Rotate On The Y
glRotatef(zrot,0.0f,0.0f,1.0f); // Rotate On The Z
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Our Texture
```

You've seen all of this before as well. Same as in tutorial #6 except I merely push my scene back away from the camera a bit more.

Merely starts the loop to draw our polygons. I use integers here to keep from having to use the int() function as I did earlier to get the array reference returned as an integer.

```
float_x = float(x)/44.0f; // Create A Floating Point 2
float_y = float(y)/44.0f; // Create A Floating Point 3
float_xb = float(x+1)/44.0f; // Create A Floatin
float_yb = float(y+1)/44.0f; // Create A Floatin
```

We use the four variables above for the texture coordinates. Each of our polygons (square in the grid), has a $1/44 \times 1/44$ section of the texture mapped on it. The loops will specify the lower left vertex and then we just add to it accordingly to get the other three (i.e. x+1 or y+1).

```
glTexCoord2f( float_x, float_y); // First Texture Coordinate
glVertex3f( points[x][y][0], points[x][y][1], points[x][y][2] );
glTexCoord2f( float_x, float_yb ); // Second Texture Coordinate
glVertex3f( points[x][y+1][0], points[x][y+1][1], points[x][y+1]
glTexCoord2f( float_xb, float_yb ); // Third Texture Coordinate
glVertex3f( points[x+1][y+1][0], points[x+1][y+1][1], points[x+1]
glTexCoord2f( float_xb, float_y ); // Fourth Texture Coordinate
glVertex3f( points[x+1][y][0], points[x+1][y][1], points[x+1][y]
}
}
glEnd(); // Done Drawing Our Quads
```

The lines above merely make the OpenGL calls to pass all the data we talked about. Four separate calls to each glTexCoord2f() and glVertex3f(). Continue with the following. Notice the quads are drawn clockwise. This means the face you see initially will be the back. The back is filled in. The front is made up of lines.

If you drew in a counter clockwise order the face you'd initially see would be the front face, meaning you would see the grid type texture instead of the filled in face.

```
if( wiggle_count == 2 ) // Used To Slow Dov
{
```

If we've drawn two scenes, then we want to cycle our sine values giving us "motion".

```
for( y = 0; y < 45; y++ )
                                                         // Loop Through The Y Plane
                   hold=points[0][y][2];
                                                                    // Store Current Va
                   hold=points[0][y][2];// Store Current Vafor( x = 0; x < 44; x++)</td>// Loop Through The X Plane
                   {
                             // Current Wave Value Equals Value To The Right
                             points[x][y][2] = points[x+1][y][2];
                   }
                   points[44][y][2]=hold;
                                                                    // Last Value Becor
         }
                                                     // Set Counter Back To Zero
         wiggle_count = 0;
                                                                    // Increase The Cou
wiggle_count++;
```

What we do here is store the first value of each line, we then move the wave to the left one, causing the image to wave. The value we stored is then added to the end to create a never ending wave across the face of the texture. Then we reset the counter **wiggle_count** to keep our animation going.

The above code was modified by NeHe (Feb 2000), to fix a flaw in the ripple going across the surface of the texture. The ripple is now smooth.

xrot+=0.3f;// Increase The X Iyrot+=0.2f;// Increase The Y Izrot+=0.4f;// Increase The Z Ireturn TRUE;// Jump Back

}

Standard NeHe rotation values. :) And that's it folks. Compile and you should have a nice rotating bitmapped "wave". I'm not sure what else to say except, whew.. This was LONG! But I hope you guys can follow it/get something out of it. If you have any questions, want me to clear something up or tell me how god awful, lol, I code, then send me a note.

This was a blast, but very energy/time consuming. It makes me appreciate the likes of NeHe ALOT more now. Thanks all.

Bosco (bosco4@home.com)

* DOWNLOAD Visual C++ Code For This Lesson.

- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Power Basic Code For This Lesson. (Conversion by Angus Law)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 12

In this tutorial I'll teach you how to use Display Lists. Not only do display lists speed up your code, they also cut down on the number of lines of code you need to write when creating a simple GL scene.

For example. Lets say you're making the game asteroids. Each level starts off with at least 2 asteroids. So you sit down with your graph paper (grin), and figure out how to make a 3D asteroid. Once you have everything figured out, you build the asteroid in OpenGL using Polygons or Quads. Lets say the asteroid is octagonal (8 sides). If you're smart you'll create a loop, and draw the asteroid once inside the loop. You'll end up with roughly 18 lines or more of code to make the asteroid. Creating the asteroid each time it's drawn to the screen is hard on your system. Once you get into more complex objects you'll see what I mean.

So what's the solution? Display Lists!!! By using a display list, you create the object just once. You can texture map it, color it, whatever you want to do. You give the display list a name. Because it's an asteroid we'll call the display list 'asteroid'. Now any time I want to draw the textured / colored asteroid on the screen, all I have to do is call glCallList(asteroid). the premade asteroid will instantly appear on the screen. Because the asteroid has already built in the display list, OpenGL doesn't have to figure out how to build it. It's prebuilt in memory. This takes alot of strain off your processor and allows your programs to run alot faster!

So are you ready to learn? :) We'll call this the Q-Bert Display List demo. What you'll end up with is a Q-Bert type screen made up of 15 cubes. Each cube is made up of a TOP, and a BOX. The top will be a seperate display list so that we can color it a darker shade. The box is a cube without the top :)

This code is based around lesson 6. I'll rewrite most of the program so it's easier to see where I've made changes. The follow lines of code are standard code used in just about all the lessons.

#include <windows.h>
#include <stdio.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>

HDC hDC=NULL; HGLRC hRC=NULL; HWND hWnd=NULL; HINSTANCEhInstance;

bool keys[256]; bool active=TRUE; bool fullscreen=TRUE; // Header File For Windows
// Header File For Standard Input/Out
// Header File For The OpenGL32 Libra
// Header File For The GLu32
// Header File For The GLaux

- // Fullscreen Flag Set To Fullscreen

Now we set up our variables. First we set up storage for one texture. Then we create two new variables for our 2 display lists. These variable will act as pointers to where the display list is stored in ram. They're called box and top.

After that we have 2 variables called xloop and yloop which are used to position the cubes on the screen and 2 variables called xrot and yrot that are used to rotate the cubes on the x axis and y axis.

GLuint	<pre>texture[1];</pre>	// Storage For One Texture
GLuint	box;	// Storage For The Display I
GLuint	top;	// Storage For The Second Di
GLuint	xloop;	// Loop For X Axis
GLuint	yloop;	// Loop For Y Axis
GLfloat	xrot;	// Rotates Cube On The X Axi
GLfloat	yrot;	// Rotates Cube On The Y Axi
	-	

Next we create two color arrays. The first one boxcol stores the color values for Bright Red, Orange, Yellow, Green and Blue. Each value inside the {}'s represent a red, green and blue value. Each group of {}'s is a specific color.

The second color array we create is for Dark Red, Dark Orange, Dark Yellow, Dark Green and Dark Blue. The dark colors will be used to draw the top of the boxes. We want the lid to be darker than the rest of the box.

Now we build the actual Display List. If you notice, all the code to build the box is in the first list, and all the code to build the top is in the other list. I'll try to explain this section in alot of detail.

```
GLvoid BuildList() // Build Box Display List {
```

We start off by telling OpenGL we want to build 2 lists. glGenLists(2) creates room for the two lists, and returns a pointer to the first list. 'box' will hold the location of the first list. Whenever we call box the first list will be drawn.

box=glGenLists(2);

// Building Two Lists

Now we're going to build the first list. We've already freed up room for two lists, and we know that box points to the area we're going to store the first list. So now all we have to do is tell OpenGL where the list should go, and what type of list to make.

We use the command glNewList() to do the job. You'll notice box is the first parameter. This tells OpenGL to store the list in the memory location that box points to. The second parameter GL_COMPILE tells OpenGL we want to prebuild the list in memory so that OpenGL doesn't have to figure out how to create the object ever time we draw it.

GL_COMPILE is similar to programming. If you write a program, and load it into your compiler, you have to compile it every time you want to run it. If it's already compiled into an .EXE file, all you have to do is click on the .exe to run it. No compiling needed. Once GL has compiled the display list, it's ready to go, no more compiling required. This is where we get the speed boost from using display lists.

glNewList(box,GL_COMPILE);

// New Compiled box Display List

The next section of code draws the box without the top. It wont appear on the screen. It will be stored in the display list.

You can put just about any command you want between glNewList() and glEndList(). You can set colors, you can change textures, etc. The only type of code you CAN'T add is code that would change the display list on the fly. Once the display list is built, you CAN'T change it.

If you added the line glColor3ub(rand()%255,rand()%255,rand()%255) into the code below, you might think that each time you draw the object to the screen it will be a different color. But because the list is only CREATED once, the color will not change each time you draw it to the screen. Whatever color the object was when it was first made is the color it will remain.

If you want to change the color of the display list, you have to change it BEFORE you draw the display list to the screen. I'll explain more on this later.

glBegin(GL_QUADS);

.0f);
• • • • / /
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);
.0f);

```
// Left Face
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
```

```
We tell OpenGL we're done making out list with the command glEndList(). Anything between
```

glNewList() and glEndList is part of the Display List, anything before glNewList() or after glEndList() is not part of the current display list.

glEndList();

Now we'll make our second display list. To find out where the second display list is stored in memory, we take the value of the old display list (box) and add one to it. The code below will make 'top' equal the location of the second display list.

top=box+1;

Now that we know where to store the second display list, we can build it. We do this the same way we built the first display list, but this time we tell OpenGL to store the list at 'top' instead of 'box'.

glNewList(top,GL_COMPILE);

The following section of code just draws the top of the box. It's a simple quad drawn on the Z plane.

Again we tell OpenGL we're done building our second list with the command glEndList(). That's it. We've successfully created 2 display lists.

glEndList();

}

The bitmap/texture building code is the same code we used in previous tutorials to load and build a texture. We want a texture that we can map onto all 6 sides of each cube. I've decided to use mipmapping to make the texture look real smooth. I hate seeing pixels :) The name of the texture to load is called 'cube.bmp'. It's stored in a directory called data. Find LoadBMP and change that line to look like the line below.

if (TextureImage[0]=LoadBMP("Data/Cube.bmp")) // Load The Bitmap

Resizing code is exactly the same as the code in Lesson 6.

The init code only has a few changes. I've added the line BuildList(). This will jump to the section of code that builds the display lists. Notice that BuildList() is after LoadGLTextures(). It's important to know the order things should go in. First we build the textures, so when we create our display lists, there's a texture already created that we can map onto the cube.

```
int InitGL(GLvoid)
                                                               // All Setup For OpenGL Goes
{
         if (!LoadGLTextures())
                                                                         // Jump To Texture
         {
                                                                         // If Texture Didn
                 return FALSE;
         }
         BuildLists();
                                                                        // Jump To The Code
         glEnable(GL_TEXTURE_2D);
                                                               // Enable Texture Mapping
         glShadeModel(GL_SMOOTH);
                                                               // Enable Smooth Shading
        glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
                                                                        // Black Background
         glClearDepth(1.0f);
                                                                        // Depth Buffer Set
         glEnable(GL_DEPTH_TEST);
                                                               // Enables Depth Testing
         glDepthFunc(GL_LEQUAL);
                                                                        // The Type Of Dept
```

The next three lines of code enable quick and dirty lighting. Light0 is predefined on most video cards, so it saves us the hassle of setting up lights. After we enable light0 we enable lighting. If light0 isn't working on your video card (you see blackness), just disable lighting.

The last line GL_COLOR_MATERIAL lets us add color to texture maps. If we don't enable material coloring, the textures will always be their original color. glColor3f(r,g,b) will have no affect on the coloring. So it's important to enable this.

glEnable(GL_LIGHT0); glEnable(GL_LIGHTING); glEnable(GL_COLOR_MATERIAL); // Quick And Dirty
// Enable Lighting
// Enable Material

Finally we set the perspective correction to look nice, and we return TRUE letting our program know that initialization went OK.

glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nice Perspective Correcti
return TRUE; // Initialization V

Now for the drawing code. As usual, I got a little crazy with the math. No SIN, and COS, but it's still a little strange :) We start off as usual by clearing the screen and depth buffer.

Then we bind a texture to the cube. I could have added this line inside the display list code, but by leaving it outside the display list, I can change the texture whenever I want. If I added the line glBindTexture(GL_TEXTURE_2D, texture[0]) inside the display list code, the display list would be built with whatever texture I selected permanently mapped onto it.

```
int DrawGLScene(GLvoid) // Here's Where We
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The
    glBindTexture(GL_TEXTURE_2D, texture[0]); // Select The Texture
```

Now for the fun stuff. We have a loop called yloop. This loop is used to position the cubes on the Y axis (up and down). We want 5 rows of cubes up and down, so we make a loop from 1 to less than 6 (which is 5).

for (yloop=1;yloop<6;yloop++)
{</pre>

// Loop Through The

We have another loop called xloop. It's used to position the cubes on the X axis (left to right). The number of cubes drawn left to right depends on what row we're on. If we're on the top row, xloop will only go from 0 to 1 (drawing one cube). the next row xloop will go from 0 to 2 (drawing 2 cubes), etc.

```
for (xloop=0;xloop;xloop++) // Loop Through The X Plane
{
```

We reset our view with glLoadIdentity().

glLoadIdentity();

// Reset The View

The next line translates to a specific spot on the screen. It looks confussing, but it's actually not. On the X axis, the following happens:

We move to the right 1.4 units so that the pyramid is in the center of the screen. Then we multiply xloop by 2.8 and add the 1.4 to it. (we multiply by 2.8 so that the cubes are not on top of eachother (2.8 is roughly the width of the cubes when they're rotated 45 degrees). Finally we subtract yloop*1.4. This moves the cubes left depending on what row we're on. If we didn't move to the left, the pyramid would line up on the left side (wouldn't really look a pyramid would it).

On the Y axis we subtract yloop from 6 otherwise the pyramid would be built upside down. Then we multiply the result by 2.4. Otherwise the cubes would be on top of eachother on the y axis (2.4 is roughly the height of each cube). Then we subtract 7 so that the pyramid starts at the bottom of the screen and is built upwards.

Finally, on the Z axis we move into the screen 20 units. That way the pyramid fits nicely on the screen.

```
// Position The Cubes On The Screen
glTranslatef(1.4f+(float(xloop)*2.8f)-(float(yloop)*1.4f),((6.0f
```

Now we rotate on the x axis. We'll tilt the cube towards the view by 45 degrees minus 2 multiplied by yloop. Perspective mode tilts the cubes automatically, so I subtract to compensate for the tilt. Not the best way to do it, but it works :)

Finally we add xrot. This gives us keyboard control over the angle. (fun to play around with).

After we've rotated on the x axis, we rotate 45 degrees on the y axis, and add yrot so we have keyboard control on the y axis.

glRotatef(45.0f-(2.0f*yloop)+xrot,1.0f,0.0f,0.0f); // Tilt T glRotatef(45.0f+yrot,0.0f,1.0f,0.0f);

Next we select a box color (bright) before we actually draw the box portion of the cube. Notice we're using glColor3fv(). What this does is loads all three values (red, green, blue) from inside the {}'s at once and sets the color. 3fv stands for 3 values, floating point, v is a pointer to an array. The color we select is yloop-1 which gives us a different color for each row of the cubes. If we used xloop-1 we'd get a different color for each column.

glColor3fv(boxcol[yloop-1]);

// Select A Box Co.

Now that the color is set, all we have to do is draw our box. Instead of writing out all the code to draw a box, all we do is call our display list. We do this with the command glCallList(box). box tells OpenGL to select the box display list. The box display list is the cube without its top.

The box will be drawn using the color we selected with glColor3fv(), at the position we translated to.

glCallList(box);

// Draw The Box

Now we select a top color (darker) before we draw the top of the box. If you actually wanted to make Q-Bert, you'd change this color whenever Q-Bert jumped on the box. The color depends on the row (yloop-1).

glColor3fv(topcol[yloop-1]);

// Select The Top (

Finally, the only thing left to do is draw the top display list. This will add a darker colored lid to the box. That's it. Very easy!

```
glCallList(top); // Draw The Top
}
return TRUE; // Jump Back
}
```

The remaining changes have all been made in WinMain(). The code has been added right after our SwapBuffers(hDC) line. It check to see if we are pressing left, right, up or down, and moves the cubes accordingly.

```
SwapBuffers(hDC);
                                               // Swap Buffers (Double Buff
if (keys[VK_LEFT])
                                               // Left Arrow Being Pressed:
{
                                                         // If So Spin Cube:
         yrot-=0.2f;
}
                                                         // Right Arrow Bein
if (keys[VK_RIGHT])
{
                                                         // If So Spin Cube:
         yrot+=0.2f;
}
if (keys[VK_UP])
                                               // Up Arrow Being Pressed?
{
                                                         // If So Tilt Cube:
         xrot-=0.2f;
}
if (keys[VK_DOWN])
                                               // Down Arrow Being Pressed:
{
                                                         // If So Tilt Cube:
         xrot+=0.2f;
}
```

Like all the previous tutorials, make sure the title at the top of the window is correct.

```
if (keys[VK_F1])
                                                               // Is F1 Being Pressed?
                  {
                           keys[VK_F1]=FALSE;
                                                              // If So Make Key FALSE
                           KillGLWindow();
                                                                        // Kill Our Current
                           fullscreen=!fullscreen;
                                                                        // Toggle Fullscre
                           // Recreate Our OpenGL Window
                           if (!CreateGLWindow("NeHe's Display List Tutorial",640,480,16,fu
                           {
                                                               // Quit If Window Was Not Ci
                                   return 0;
                           }
                  }
        }
}
```

By the end of this tutorial you should have a good understanding of how display lists work, how to create them, and how to display them on the screen. Display lists are great. Not only do they simplify coding complex projects, they also give you that little bit of extra speed required to maintain high framerates.

I hope you've enjoy the tutorial. If you have any questions or feel somethings not clear, please email me and let me know.

Jeff Molofee (NeHe)

Jeff Molofee's OpenGL Windows Tutorial #12

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts) * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)

Back To NeHe Productions!

Lesson 13

Welcome to yet another Tutorial. This time on I'll be teaching you how to use Bitmap Fonts. You may be saying to yourself "what's so hard about putting text onto the screen". If you've ever tried it, it's not that easy!

Sure you can load up an art program, write text onto an image, load the image into your OpenGL program, turn on blending then map the text onto the screen. But this is time consuming, the final result usually looks blurry or blocky depending on the type of filtering you use, and unless your image has an alpha channel your text will end up transparent (blended with the objects on the screen) once it's mapped to the screen.

If you've ever used Wordpad, Microsoft Word or some other Word Processor, you may have noticed all the different types of Font's available. This tutorial will teach you how to use the exact same fonts in your own OpenGL programs. As a matter of fact... Any font you install on your computer can be used in your demos.

Not only do Bitmap Fonts looks 100 times better than graphical fonts (textures). You can change the text on the fly. No need to make textures for each word or letter you want to write to the screen. Just position the text, and use my handy new gl command to display the text on the screen.

I tried to make the command as simple as possible. All you do is type glPrint("Hello"). It's that easy. Anyways. You can tell by the long intro that I'm pretty happy with this tutorial. It took me roughly 1 1/2 hours to create the program. Why so long? Because there is literally no information available on using Bitmap Fonts, unless of course you enjoy MFC code. In order to keep the code simple I decided it would be nice if I wrote it all in simple to understand C code :)

A small note, this code is Windows specific. It uses the wgl functions of Windows to build the font. Apparently Apple has agl support that should do the same thing, and X has glx. Unfortunately I can't guarantee this code is portable. If anyone has platform independant code to draw fonts to the screen, send it my way and I'll write another font tutorial.

We start off with the typical code from lesson 1. We'll be adding the stdio.h header file for standard input/output operations; the stdarg.h header file to parse the text and convert variables to text, and finally the math.h header file so we can move the text around the screen using SIN and COS.

```
#include <windows.h> // Header File For Windows
#include <math.h> // Header File For Windows Math Library ( ADD )
#include <stdio.h> // Header File For Standard Input/Output ( ADD )
#include <stdarg.h> // Header File For Variable Argument Routines ( ADD )
#include <gl\gl.h> // Header File For The OpenGL32 Library
#include <gl\glu.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Holds Our Window Handle
#INSTANCEhInstance; // Holds The Instance Of The Application
```

We're going to add 3 new variables as well. **base** will hold the number of the first display list we create. Each character requires it's own display list. The character 'A' is 65 in the display list, 'B' is 66, 'C' is 67, etc. So 'A' would be stored in display list **base**+65.

Next we add two counters (**cnt1** & **cnt2**). These counters will count up at different rates, and are used to move the text around the screen using SIN and COS. This creates a semi-random looking movement on the screen. We'll also use the counters to control the color of the letters (more on this later).

```
GLuint base; // Base Display List For The Font Set

GLfloat cnt1; // 1st Counter Used To Move Text & For Coloring

GLfloat cnt2; // 2nd Counter Used To Move Text & For Coloring

bool keys[256]; // Array Used For The Keyboard Routine

bool active=TRUE; // Window Active Flag Set To TRUE By Default

bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc
```

The following section of code builds the actual font. This was the most difficult part of the code to write. 'HFONT font' in simple english tells Windows we are going to be manipulating a Windows font.

Next we define **base**. We do this by creating a group of 96 display lists using glGenLists(96). After the display lists are created, the variable **base** will hold the number of the first list.

GLvoid BuildFont(GLvoid)	// Build Our Bitmap Font
HFONT font;	// Windows Font ID
<pre>base = glGenLists(96);</pre>	// Storage For 96 (

Now for the fun stuff. We're going to create our font. We start off by specifying the size of the font. You'll notice it's a negative number. By putting a minus, we're telling windows to find us a font based on the CHARACTER height. If we use a positive number we match the font based on the CELL height.

font = CreateFont(-24,

// Height Of Font (NEW)

Then we specify the cell width. You'll notice I have it set to 0. By setting values to 0, windows will use the default value. You can play around with this value if you want. Make the font wide, etc.

Ο,

// Width Of Font

value, the thicker the font (more bold).

Angle of Escapement will rotate the font. Unfortunately this isn't a very useful feature. Unless your at 0, 90, 180, and 270 degrees, the font usually gets cropped to fit inside it's invisible square border. Orientation Angle quoted from MSDN help Specifies the angle, in tenths of degrees, between each character's base line and the x-axis of the device. Unfortunately I have no idea what that means :(

Font weight is a great parameter. You can put a number from 0 - 1000 or you can use one of the predefined values. FW_DONTCARE is 0, FW_NORMAL is 400, FW_BOLD is 700 and FW_BLACK is 900. There are alot more predefined values, but those 4 give some good variety. The higher the

FW_BOLD,

Ο,

Ο,

// Font Weight

// Angle Of Escape

// Orientation Ang

Italic, Underline and Strikeout can be either TRUE or FALSE. Basically if underline is TRUE, the font will be underlined. If it's FALSE it wont be. Pretty simple :)

FALSE,	11	Italic
FALSE,	11	Underline
FALSE,	11	Strikeout

Character set Identifier describes the type of Character set you wish to use. There are too many types to explain. CHINESEBIG5_CHARSET, GREEK_CHARSET, RUSSIAN_CHARSET, DEFAULT_CHARSET, etc. ANSI is the one I use, although DEFAULT would probably work just as well.

If you're interested in using a font such as Webdings or Wingdings, you need to use SYMBOL_CHARSET instead of ANSI_CHARSET.

ANSI_CHARSET,

// Character Set Id

Output Precision is very important. It tells Windows what type of character set to use if there is more than one type available. OUT_TT_PRECIS tells Windows that if there is more than one type of font to choose from with the same name, select the TRUETYPE version of the font. Truetype fonts always look better, especially when you make them large. You can also use OUT_TT_ONLY_PRECIS, which ALWAYS trys to use a TRUETYPE Font.

OUT_TT_PRECIS,

// Output Precision

Clipping Precision is the type of clipping to do on the font if it goes outside the clipping region. Not much to say about this, just leave it set to default.

CLIP_DEFAULT_PRECIS,

// Clipping Precis:

Output Quality is very important.you can have PROOF, DRAFT, NONANTIALIASED, DEFAULT or ANTIALIASED. We all know that ANTIALIASED fonts look good :) Antialiasing a font is the same effect you get when you turn on font smoothing in Windows. It makes everything look less jagged.

ANTIALIASED_QUALITY,

// Output Quality

Next we have the Family and Pitch settings. For pitch you can have DEFAULT_PITCH, FIXED_PITCH and VARIABLE_PITCH, and for family you can have FF_DECORATIVE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS, FF_DONTCARE. Play around with them to find out what they do. I just set them both to default.

FF_DONTCARE DEFAULT_PITCH, // Family And Pitch

Finally... We have the actual name of the font. Boot up Microsoft Word or some other text editor. Click on the font drop down menu, and find a font you like. To use the font, replace 'Courier New' with the name of the font you'd rather use.

"Courier New");

// Font Name

Now we select the font by relating it to our DC, and build the 96 display lists starting at character 32 (which is a blank space). You can build all 256 if you'd like, just make sure you build 256 display lists using glGenLists. Make sure you delete all 256 display lists when you quit the program, and make sure you set 32 to 0 and 96 to 255 in the line below.

SelectObject(hDC, font);// Selects The Font We CreatwglUseFontBitmaps(hDC, 32, 96, base);// Builds 96 Charac

}

The following code is pretty simple. It deletes the 96 display lists from memory starting at the first list specified by 'base'. I'm not sure if windows would do this for you, but it's better to be safe than sorry :)

GLvoid KillFont(GLvoid)
{
 glDeleteLists(base, 96);
}

// Delete The Font
// Delete All 96 Characters

Now for my handy dandy GL text routine. You call this section of code with the command glPrint ("message goes here"). The text is stored in the char string *fmt.

The next two lines of code check to see if there's anything to display? If there's no text, fmt will equal nothing (NULL), and nothing will be drawn to the screen.

The following three lines of code convert any symbols in the text to the actual numbers the symbols represent. The final text and any converted symbols are then stored in the character string called "text". I'll explain symbols in more detail down below.

<pre>va_start(ap, fmt);</pre>			
vsprintf(text,	fmt,	ap);	
<pre>va_end(ap);</pre>			

We then push the GL_LIST_BIT, this prevents glListBase from affecting any other display lists we may be using in our program.

The command glListBase(base-32) is a little hard to explain. Say we draw the letter 'A', it's represented by the number 65. Without glListBase(base-32) OpenGL wouldn't know where to find this letter. It would look for it at display list 65, but if base was equal to 1000, 'A' would actually be stored at display list 1065. So by setting a base starting point, OpenGL knows where to get the proper display list from. The reason we subtract 32 is because we never made the first 32 display lists. We skipped them. So we have to let OpenGL know this by subtracting 32 from the base value. I hope that makes sense.

glPushAttrib(GL_LIST_BIT);
glListBase(base - 32);

// Pushes The Display List H // Sets The Base Cl Now that OpenGL knows where the Letters are located, we can tell it to write the text to the screen. glCallLists is a very interesting command. It's capable of putting more than one display list on the screen at a time.

The line below does the following. First it tells OpenGL we're going to be displaying lists to the screen. strlen(text) finds out how many letters we're going to send to the screen. Next it needs to know what the largest list number were sending to it is going to be. We're not sending any more than 255 characters. So we can use an UNSIGNED_BYTE. (remember a byte is any value from 0 - 255). Finally we tell it what to display by passing the string 'text'.

In case you're wondering why the letters don't pile on top of eachother. Each display list for each character knows where the right side of the letter is. After the letter is drawn, OpenGL translates to the right side of the drawn letter. The next letter or object drawn will be drawn starting at the last location GL translated to, which is to the right of the last letter.

Finally we pop the GL_LIST_BIT setting GL back to how it was before we set our base setting using glListBase(base-32).

```
glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // Draws The Display List Te
glPopAttrib(); // Pops The Display
```

}

The only thing different in the Init code is the line BuildFont(). This jumps to the code above that builds the font so OpenGL can use it later on.

```
int InitGL(GLvoid)
                                                               // All Setup For OpenGL Goes
{
        glShadeModel(GL_SMOOTH);
                                                               // Enable Smooth Shading
        glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
                                                                        // Black Background
        glClearDepth(1.0f);
                                                                        // Depth Buffer Set
        glEnable(GL_DEPTH_TEST);
                                                               // Enables Depth Testing
                                                                        // The Type Of Dept
        glDepthFunc(GL_LEQUAL);
        glHint(GL PERSPECTIVE CORRECTION HINT, GL NICEST); // Really Nice Perspective (
        BuildFont();
                                                                        // Build The Font
        return TRUE;
                                                                        // Initialization I
```

}

Now for the drawing code. We start off by clearing the screen and the depth buffer. We call glLoadIdentity() to reset everything. Then we translate one unit into the screen. If we don't translate, the text wont show up. Bitmap fonts work better when you use an ortho projection rather than a perspective projection, but ortho looks bad, so to make it work in projection, translate.

You'll notice that if you translate even deeper into the screen the size of the font does not shrink like you'd expect it to. What actually happens when you translate deeper is that you have more control over where the text is on the screen. If you translate 1 unit into the screen, you can place the text anywhere from -0.5 to +0.5 on the X axis. If you tranlate 10 units into the screen, you place the text from -5 to +5. It just gives you more control instead of using decimal places to position the text at exact locations. Nothing will change the size of the text. Not even glScalef(x,y,z). If you want the font bigger or smaller, make it bigger or smaller when you create it!

```
int DrawGLScene(GLvoid) // Here's Where We
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The
    glLoadIdentity(); // Reset The View
    glTranslatef(0.0f,0.0f,-1.0f); // Move One Unit In
```

Now we use some fancy math to make the colors pulse. Don't worry if you don't understand what I'm doing. I like to take advantage of as many variables and stupid tricks as I can to achieve results :)

In this case I'm using the two counters we made to move the text around the screen to change the red, green and blue colors. Red will go from -1.0 to 1.0 using COS and counter 1. Green will also go from -1.0 to 1.0 using SIN and counter 2. Blue will go from 0.5 to 1.5 using COS and counter 1 and 2. That way blue will never be 0, and the text should never completely fade out. Stupid, but it works :)

```
// Pulsing Colors Based On Text Position
glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt))
```

Now for a new command. glRasterPos2f(x,y) will position the Bitmapped Font on the screen. The center of the screen is still 0,0. Notice there's no Z position. Bitmap Fonts only use the X axis (left/right) and Y axis (up/down). Because we translate one unit into the screen, the far left is -0.5, and the far right is +0.5. You'll notice that I move 0.45 pixels to the left on the X axis. This moves the text into the center of the screen. Otherwise it would be more to the right of the screen because it would be drawn from the center to the right.

The fancy(?) math does pretty much the same thing as the color setting math does. It moves the text on the x axis from -0.50 to -0.40 (remember, we subtract 0.45 right off the start). This keeps the text on the screen at all times. It swings left and right using COS and counter 1. It moves from - 0.35 to +0.35 on the Y axis using SIN and counter 2.

// Position The Text On The Screen
glRasterPos2f(-0.45f+0.05f*float(cos(cnt1)), 0.35f*float(sin(cnt2)));

Now for my favorite part... Writing the actual text to the screen. I tried to make it super easy, and very user friendly. You'll notice it looks alot like an OpenGL call, combined with the good old fashioned Print statement :) All you do to write the text to the screen is glPrint("{any text you want}"). It's that easy. The text will be drawn onto the screen at the exact spot you positioned it.

Shawn T. sent me modified code that allows glPrint to pass variables to the screen. This means that you can increase a counter and display the results on the screen! It works like this... In the line below you see our normal text. Then there's a space, a dash, a space, then a "symbol" (%7.2f). Now you may look at %7.2f and say what the heck does that mean. It's very simple. % is like a marker saying don't print 7.2f to the screen, because it represents a variable. The 7 means a maximum of 7 digits will be displayed to the left of the decimal place. Then the decimal place, and right after the decimal place is a 2. The 2 means that only two digits will be displayed to the right of the decimal place. Finally, the f. The f means that the number we want to display is a floating point number. We want to display the value of **cnt1** on the screen would be 300.12. The 3, 4, and 5 after the decimal place would be cut off because we only want 2 digits to appear after the decimal place.

I know if you're an experienced C programmer, this is absolute basic stuff, but there may be people

out there that have never used printf. If you're interested in learning more about symbols, buy a book, or read through the MSDN.

```
glPrint("Active OpenGL Text With NeHe - %7.2f", cntl); // Print GL Text To The Scre
```

The last thing to do is increase both the counters by different amounts so the colors pulse and the text moves.

```
cnt1+=0.051f;
cnt2+=0.005f;
return TRUE;
```

// Increase The Fin
// Increase The Sec
// Everything Went

}

}

The last thing to do is add KillFont() to the end of KillGLWindow() just like I'm showing below. It's important to add this line. It cleans things up before we exit our program.

```
if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To Unregister
{
     MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | MB
     hInstance=NULL; // Set hInstance Tc
}
KillFont(); // Destroy The Font
```

That's it... Everything you need to know in order to use Bitmap Fonts in your own OpenGL projects. I've searched the net looking for a tutorial similar to this one, and have found nothing. Perhaps my site is the first to cover this topic in easy to understand C code? Anyways. Enjoy the tutorial, and happy coding!

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Linux Code For This Lesson. (Conversion by Richard Campbell)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)

Back To NeHe Productions!

Lesson 14

This tutorial is a sequel to the last tutorial. In tutorial 13 I taught you how to use Bitmap Fonts. In this tutorial I'll teach you how to use Outline Fonts.

The way we create Outline fonts is fairly similar to the way we made the Bitmap font in lesson 13. However... Outline fonts are about 100 times more cool! You can size Outline fonts. Outline font's can move around the screen in 3D, and outline fonts can have thickness! No more flat 2D characters. With Outline fonts, you can turn any font installed on your computer into a 3D font for OpenGL, complete with proper normals so the characters light up really nice when light shines on them.

A small note, this code is Windows specific. It uses the wgl functions of Windows to build the font. Apparently Apple has agl support that should do the same thing, and X has glx. Unfortunately I can't guarantee this code is portable. If anyone has platform independant code to draw fonts to the screen, send it my way and I'll write another font tutorial.

We start off with the typical code from lesson 1. We'll be adding the stdio.h header file for standard input/output operations; the stdarg.h header file to parse the text and convert variables to text, and finally the math.h header file so we can move the text around the screen using SIN and COS.

<pre>#include <windows.h></windows.h></pre>	// Header File For Windows	
<pre>#include <math.h></math.h></pre>	// Header File For Windows Math Library	(ADD)
#include <stdio.h></stdio.h>	// Header File For Standard Input/Output (ADD)	
<pre>#include <stdarg.h></stdarg.h></pre>	// Header File For Variable Argument Routines	(ADD)
<pre>#include <gl\gl.h></gl\gl.h></pre>	// Header File For The OpenGL32 Library	
<pre>#include <gl\glu.h></gl\glu.h></pre>	// Header File For The GLu32 Library	
<pre>#include <gl\glaux.h></gl\glaux.h></pre>	// Header File For The GLaux Library	
HDC hDC=NULL	;// Private GDI Device Context	
HGLRC hRC=NULL	;// Permanent Rendering Context	
HWND hWnd=NUL	L; // Holds Our Window Handle	
HINSTANCEhInstance; // Holds The Instance Of The Application		

We're going to add 2 new variables. **base** will hold the number of the first display list we create. Each character requires it's own display list. The character 'A' is 65 in the display list, 'B' is 66, 'C' is 67, etc. So 'A' would be stored in display list **base**+65.

Next we add a variable called **rot**. **rot** will be used to spin the text around on the screen using both SIN and COS. It will also be used to pulse the colors.

GLuint	base;	// Base Display List For The Font Set (ADD)	
GLfloat	rot;	// Used To Rotate The Text (ADD)	
bool	keys[256];	// Array Used For The Keyboard Routine	
bool	active=TRUE;	// Window Active Flag Set To TRUE By Default	
bool	fullscreen=TRUE;	// Fullscreen Flag Set To Fullscreen Mode By Default	

GLYPHMETRICSFLOAT **gmf[256]** will hold information about the placement and orientation for each of our 256 outline font display lists. We select a letter by using **gmf**[num]. num is the number of the display list we want to know something about. Later in the code I'll show you how to find out the width of each character so that you can automatically center the text on the screen. Keep in mind that each character can be a different width. glyphmetrics will make our lives a whole lot easier.

```
GLYPHMETRICSFLOAT gmf[256]; // Storage For Information About Our Font
```

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc

The following section of code builds the actual font similar to the way we made our Bitmap font. Just like in lesson 13, this section of code was the hardest part for me to figure out.

'HFONT font' will hold our Windows font ID.

Next we define **base**. We do this by creating a group of 256 display lists using glGenLists(256). After the display lists are created, the variable **base** will hold the number of the first list.

GLvoid BuildFont(GLvoid)	// Build Our Bitmap Font
HFONT font;	// Windows Font ID
<pre>base = glGenLists(256);</pre>	// Storage For 256

More fun stuff. We're going to create our Outline font. We start off by specifying the size of the font. You'll notice it's a negative number. By putting a minus, we're telling windows to find us a font based on the CHARACTER height. If we use a positive number we match the font based on the CELL height.

font = CreateFont(-12,

// Height Of Font

Then we specify the cell width. You'll notice I have it set to 0. By setting values to 0, windows will use the default value. You can play around with this value if you want. Make the font wide, etc.

Ο,

// Width Of Font

Angle of Escapement will rotate the font. Orientation Angle quoted from MSDN help Specifies the angle, in tenths of degrees, between each character's base line and the x-axis of the device. Unfortunately I have no idea what that means :(

0,	// Angle Of Escaper
Ο,	// Orientation Ang

Font weight is a great parameter. You can put a number from 0 - 1000 or you can use one of the predefined values. FW_DONTCARE is 0, FW_NORMAL is 400, FW_BOLD is 700 and FW_BLACK is 900. There are alot more predefined values, but those 4 give some good variety. The higher the value, the thicker the font (more bold).

FW_BOLD, // Font Weight

Italic, Underline and Strikeout can be either TRUE or FALSE. Basically if underline is TRUE, the font will be underlined. If it's FALSE it wont be. Pretty simple :)

FALSE,	11	Italic
FALSE,	11	Underline
FALSE,	//	Strikeout

Character set Identifier describes the type of Character set you wish to use. There are too many types to explain. CHINESEBIG5_CHARSET, GREEK_CHARSET, RUSSIAN_CHARSET, DEFAULT_CHARSET, etc. ANSI is the one I use, although DEFAULT would probably work just as well.

If you're interested in using a font such as Webdings or Wingdings, you need to use SYMBOL_CHARSET instead of ANSI_CHARSET.

ANSI_CHARSET,

// Character Set Ic

Output Precision is very important. It tells Windows what type of character set to use if there is more than one type available. OUT_TT_PRECIS tells Windows that if there is more than one type of font to choose from with the same name, select the TRUETYPE version of the font. Truetype fonts always look better, especially when you make them large. You can also use OUT_TT_ONLY_PRECIS, which ALWAYS trys to use a TRUETYPE Font.

OUT_TT_PRECIS,

// Output Precision

Clipping Precision is the type of clipping to do on the font if it goes outside the clipping region. Not much to say about this, just leave it set to default.

CLIP_DEFAULT_PRECIS,

Output Quality is very important.you can have PROOF, DRAFT, NONANTIALIASED, DEFAULT or ANTIALIASED. We all know that ANTIALIASED fonts look good :) Antialiasing a font is the same effect you get when you turn on font smoothing in Windows. It makes everything look less jagged.

// Output Quality

// Clipping Precis:

Next we have the Family and Pitch settings. For pitch you can have DEFAULT_PITCH, FIXED_PITCH and VARIABLE_PITCH, and for family you can have FF_DECORATIVE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS, FF_DONTCARE. Play around with them to find out what they do. I just set them both to default.

```
FF_DONTCARE DEFAULT_PITCH, // Family And Pitch
```

Finally... We have the actual name of the font. Boot up Microsoft Word or some other text editor. Click on the font drop down menu, and find a font you like. To use the font, replace 'Comic Sans MS' with the name of the font you'd rather use.

"Comic Sans MS"); // Font Name

Now we select the font by relating it to our DC.

SelectObject(hDC, font);

// Selects The Font We Creat

Now for the new code. We build our Outline font using a new command wglUseFontOutlines. We select our DC, the starting character, the number of characters to create and the 'base' display list value. All very similar to the way we built our Bitmap font.

wglUseFontOutlines(hDC,	// Select The Curre
	Ο,	// Starting Charact
	255,	// Number Of Displ;
	base,	// Starting Display
	/	

That's not all however. We then set the deviation level. The closer to 0.0f, the smooth the font will look. After we set the deviation, we get to set the font thickness. This describes how thick the font is on the Z axis. 0.0f will produce a flat 2D looking font and 1.0f will produce a font with some depth.

The parameter WGL_FONT_POLYGONS tells OpenGL to create a solid font using polygons. If we use WGL_FONT_LINES instead, the font will be wireframe (made of lines). It's also important to note that if you use GL_FONT_LINES, normals will not be generated so lighting will not work properly.

The last parameter gmf points to the address buffer for the display list data.

0.0f, // Deviation From 5 0.2f, // Font Thickness 5 WGL_FONT_POLYGONS, // Use Polygons, Not Lines gmf); // Address Of Buffe

}

The following code is pretty simple. It deletes the 256 display lists from memory starting at the first list specified by **base**. I'm not sure if Windows would do this for you, but it's better to be safe than sorry :)

```
GLvoid KillFont(GLvoid) // Delete The Font
{
    glDeleteLists(base, 256); // Delete All 256 Characters
}
```

Now for my handy dandy GL text routine. You call this section of code with the command glPrint ("message goes here"). Exactly the same way you drew Bitmap fonts to the screen in lesson 13. The text is stored in the char string fmt.

```
GLvoid glPrint(const char *fmt, ...) {
```

```
// Custom GL "Print" Routine
```

The first line below sets up a variable called **length**. We'll use this variable to find out how our string of text is. The second line creates storage space for a 256 character string. **text** is the string we will end up printing to the screen. The third line creates a pointer that points to the list of arguments we pass along with the string. If we send any variables along with the text, this pointer will point to them.

float	length=0;	// Used To Find The Length (
char	text[256];	// Holds Our String
va_list	ap;	// Pointer To List

The next two lines of code check to see if there's anything to display? If there's no text, fmt will equal nothing (NULL), and nothing will be drawn to the screen.

if (fmt == NULL)
 return;

// If There's No Text // Do Nothing

The following three lines of code convert any symbols in the text to the actual numbers the symbols represent. The final text and any converted symbols are then stored in the character string called "text". I'll explain symbols in more detail down below.

```
va_start(ap, fmt);
    vsprintf(text, fmt, ap);
va_end(ap);
```

Thanks to Jim Williams for suggesting the code below. I was centering the text manually. His method works alot better :)

We start off by making a loop that goes through all the text character by character. strlen(text) gives us the length of our text. After we've set up the loop, we will increase the value of **length** by the width of each character. When we are done the value stored in **length** will be the width of our entire string. So if we were printing "hello" and by some fluke each character was exactly 10 units wide, we'd increase the value of **length** by the width of the first letter 10. Then we'd check the width of the second letter. The width would also be 10, so **length** would become 10+10 (20). By the time we were done checking all 4 letters **length** would equal 40 (4*10).

The code that gives us the width of each character is gmf[text[loop]].gmfCellIncX. remember that gmf stores information out each display list. If **loop** is equal to 0 **text[loop]** will be the first character in our string. If **loop** is equal to 1 **text[loop]** will be the second character in our string. gmfCellIncX tells us how wide the selected character is. gmfCellIncX is actually the distance that our display moves to the right after the character has been drawn so that each character isn't drawn on top of eachother. Just so happens that distance is our width :) You can also find out the character height with the command gmfCellIncY. This might come in handy if you're drawing text vertically on the screen instead of horizontally.

Finally we take the length that we calculate and make it a negative number (because we have to move left of center to center our text). We then divide the length by 2. We don't want all the text to move left of center, just half the text!

```
glTranslatef(-length/2,0.0f,0.0f);
```

// Center Our Text On The Sc

We then push the GL_LIST_BIT, this prevents glListBase from affecting any other display lists we may be using in our program.

The command glListBase(base) tells OpenGL where to find the proper display list for each character.

glPushAttrib(GL_LIST_BIT);
glListBase(base);

// Pushes The Display List F
// Sets The Base Character t

Now that OpenGL knows where the characters are located, we can tell it to write the text to the screen. glCallLists writes the entire string of text to the screen at once by making multiple display list calls for you.

The line below does the following. First it tells OpenGL we're going to be displaying lists to the screen. strlen(**text**) finds out how many letters we're going to send to the screen. Next it needs to know what the largest list number were sending to it is going to be. We're still not sending any more than 255 characters. So we can use an UNSIGNED_BYTE. (a byte represents a number from 0 - 255 which is exactly what we need). Finally we tell it what to display by passing the string **text**.

In case you're wondering why the letters don't pile on top of eachother. Each display list for each character knows where the right side of the character is. After the letter is drawn to the screen, OpenGL translates to the right side of the drawn letter. The next letter or object drawn will be drawn starting at the last location GL translated to, which is to the right of the last letter.

Finally we pop the GL_LIST_BIT setting GL back to how it was before we set our base setting using glListBase(base).

```
glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // Draws The Display List Te
glPopAttrib();
                                                              // Pops The Display
```

}

Resizing code is exactly the same as the code in Lesson 1 so we'll skip over it.

There are a few new lines at the end of the InitGL code. The line BuildFont() from lesson 13 is still there, along with new code to do guick and dirty lighting. Light0 is predefined on most video cards and will light up the scene nicely with no effort on my part :)

I've also added the command glEnable(GL_Color_Material). Because the characters are 3D objects you need to enable Material Coloring, otherwise changing the color with glColor3f(r,g,b) will not change the color of the text. If you're drawing shapes of your own to the screen while you write text enable material coloring before you write the text, and disable it after you've drawn the text, otherwise all the object on your screen will be colored.

```
int InitGL(GLvoid)
                                                               // All Setup For OpenGL Goes
ł
        glShadeModel(GL_SMOOTH);
                                                               // Enable Smooth Shading
        glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
                                                                        // Black Background
        glClearDepth(1.0f);
                                                                        // Depth Buffer Set
        glEnable(GL_DEPTH_TEST);
                                                               // Enables Depth Testing
        glDepthFunc(GL_LEQUAL);
                                                                       // The Type Of Dept
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective (
        glEnable(GL_LIGHT0);
                                                                        // Enable Default 1
        glEnable(GL_LIGHTING);
                                                                        // Enable Lighting
        glEnable(GL_COLOR_MATERIAL);
                                                                        // Enable Coloring
        BuildFont();
                                                                        // Build The Font
        return TRUE;
                                                                        // Initialization [
}
```

Now for the drawing code. We start off by clearing the screen and the depth buffer. We call glLoadIdentity() to reset everything. Then we translate ten units into the screen. Outline fonts look great in perspective mode. The further into the screen you translate, the smaller the font becomes. The closer you translate, the larger the font becomes.

Outline fonts can also be manipulated by using the glScalef(x,y,z) command. If you want the font 2 times taller, use glScalef(1.0f,2.0f,1.0f). the 2.0f is on the y axis, which tells OpenGL to draw the list twice as tall. If the 2.0f was on the x axis, the character would be twice as wide.

```
int DrawGLScene(GLvoid)
                                                                        // Here's Where We
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The
        glLoadIdentity();
                                                               // Reset The View
```

```
glTranslatef(0.0f,0.0f,-10.0f);
```

After we've translated into the screen, we want the text to spin. The next 3 lines rotate the screen on all three axes. I multiply **rot** by different numbers to make each rotation happen at a different speed.

 glRotatef(rot,1.0f,0.0f,0.0f);
 // Rotate On The X

 glRotatef(rot*1.5f,0.0f,1.0f,0.0f);
 // Rotate On The Y Axis

 glRotatef(rot*1.4f,0.0f,0.0f,1.0f);
 // Rotate On The Z Axis

Now for the crazy color cycling. As usual, I make use of the only variable that counts up (**rot**). The colors pulse up and down using COS and SIN. I divide the value of **rot** by different numbers so that each color isn't increasing at the same speed. The final results are nice.

```
// Pulsing Colors Based On The Rotation
glColor3f(1.0f*float(cos(rot/20.0f)),1.0f*float(sin(rot/25.0f)),1.0f-0.5f*float(cos
```

My favorite part... Writing the text to the screen. I've used the same command we used to write Bitmap fonts to the screen. All you have to do to write the text to the screen is glPrint("{any text you want}"). It's that easy!

In the code below we'll print NeHe, a space, a dash, a space, and then whatever number is stored in **rot** divided by 50 (to slow down the counter a bit). If the number is larger that 999.99 the 4th digit to the left will be cut off (we're requesting only 3 digits to the left of the decimal place). Only 2 digits will be displayed after the decimal place.

glPrint("NeHe - %3.2f",rot/50);

// Print GL Text To

Then we increase the rotation variable so the colors pulse and the text spins.

```
rot+=0.5f;
return TRUE;
```

}

}

// Increase The Rot
// Everything Went

The last thing to do is add KillFont() to the end of KillGLWindow() just like I'm showing below. It's important to add this line. It cleans things up before we exit our program.

```
if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To Unregister
{
     MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | ME
     hInstance=NULL; // Set hInstance Tc
}
KillFont(); // Destroy The Font
```

At the end of this tutorial you should be able to use Outline Fonts in your own OpenGL projects. Just like lesson 13, I've searched the net looking for a tutorial similar to this one, and have found nothing. Could my site be the first to cover this topic in great detail while explaining everything in easy to understand C code? Enjoy the tutorial, and happy coding!

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)

Back To NeHe Productions!

Lesson 15

After posting the last two tutorials on bitmap and outlined fonts, I received quite a few emails from people wondering how they could texture map the fonts. You can use autotexture coordinate generation. This will generate texture coordinates for each of the polygons on the font.

A small note, this code is Windows specific. It uses the wgl functions of Windows to build the font. Apparently Apple has agl support that should do the same thing, and X has glx. Unfortunately I can't guarantee this code is portable. If anyone has platform independant code to draw fonts to the screen, send it my way and I'll write another font tutorial.

We'll build our Texture Font demo using the code from lesson 14. If any of the code has changed in a particular section of the program, I'll rewrite the entire section of code so that it's easier to see the changes that I have made.

The following section of code is similar to the code in lesson 14, but this time we're not going to include the stdarg.h file.

<pre>#include #include #include #include</pre>	<pre>e <windows.h> e <math.h> e <stdio.h> e <gl\gl.h> e <gl\glu.h> e <gl\glaux.h></gl\glaux.h></gl\glu.h></gl\gl.h></stdio.h></math.h></windows.h></pre>	<pre>// Header File For Windows // Header File For Windows Math Libra // Header File For Standard Input/Out // Header File For The OpenGL32 Libra // Header File For The GLaux // Header File For The GLaux</pre>
HDC	hDC=NULL;	// Private GDI Device Context
HGLRC	hRC=NULL;	// Permanent Rendering Context
HWND	hWnd=NULL;	// Holds Our Window Handle
HINSTANC	CEhInstance;	// Holds The Instance Of The Applicat
bool	keys[256];	// Array Used For The Keyboa
bool	active=TRUE;	// Window Active Flag Set To
bool	fullscreen=TRUE;	// Fullscreen Flag Set To Fullscreen

We're going to add one new integer variable here called **texture[]**. It will be used to store our texture. The last three lines were in tutorial 14 and have not changed in this tutorial.

 GLuint
 texture[1];
 // One Texture Map (NEW)

 GLuint
 base;
 // Base Display List For The

 GLfloat
 rot;
 // Used To Rotate The Text

 LRESULT
 CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc

The following section of code has some minor changes. In this tutorial I'm going to use the wingdings font to display a skull and crossbones type object. If you want to display text instead, you can leave the code the same as it was in lesson 14, or change to a font of your own.

A few of you were wondering how to use the wingdings font, which is another reason I'm not using a standard font. Wingdings is a SYMBOL font, and requires a few changes to make it work. It's not as easy as telling Windows to use the wingdings font. If you change the font name to wingdings, you'll notice that the font doesn't get selected. You have to tell Windows that the font is a symbol font and not a standard character font. More on this later.

GLvoid BuildFont(GLvoid)		// Build Our Bitmap Font
GLYPHMETRICSFLOAT gmf[29 HFONT font;	56];	// Address Buffer For Font Storage // Windows Font ID
<pre>base = glGenLists(256);</pre>		// Storage For 256
font = $CreateFont(-12)$		// Height Of Font
	Ο,	// Width Of Font
	Ο,	// Angle Of Escaper
	Ο,	// Orientation Ang
	FW_BOLD,	// Font Weight
	FALSE,	// Italic
	FALSE,	// Underline
	FALSE,	// Strikeout

This is the magic line! Instead of using ANSI_CHARSET like we did in tutorial 14, we're going to use SYMBOL_CHARSET. This tells Windows that the font we are building is not your typical font made up of characters. A symbol font is usually made up of tiny pictures (symbols). If you forget to change this line, wingdings, webdings and any other symbol font you may be trying to use will not work.

SYMBOL_CHARSET,

// Character Set Id

The next few lines have not changed.

OUT_TT_PRECIS,// Output PrecisionCLIP_DEFAULT_PRECIS,// Clipping Precis:ANTIALIASED_QUALITY,// Output QualityFF_DONTCARE | DEFAULT_PITCH,// Family And Pitch

Now that we've selected the symbol character set identifier, we can select the wingdings font!

"Wingdings");

// Font Name (Mod:

The remaining lines of code have not changed.

<pre>SelectObject(hDC, font);</pre>	// Selects The Font We Creat
wglUseFontOutlines(hDC, 0, 255, base	// Starting Charact // Number Of Displ;

I'm allowing for more deviation. This means GL will not try to follow the outline of the font as closely. If you set deviation to 0.0f, you'll notice problems with the texturing on really curved surfaces. If you allow for some deviation, most of the problems will disappear.

0.1f,

// Deviation From 1

The next three lines of code are still the same.

0.2f,		// 1
WGL_FONT_POLYGONS,	// Use	Polygo
gmf);		// 1

// Font Thickness :
/ Use Polygons, Not Lines
 // Address Of Buff(

}

Right before ReSizeGLScene() we're going to add the following section of code to load our texture. You might recognize the code from previous tutorials. We create storage for the bitmap image. We load the bitmap image. We tell OpenGL to generate 1 texture, and we store this texture in **texture [0]**.

I'm creating a mipmapped texture only because it looks better. The name of the texture is lights.bmp.

AUX_RGBImageRec *LoadBMP(char *Filename) {		// Loads A Bitmap Image	
t	FILE *File=NULL;	// File Handle	
	if (!Filename) {	// Make Sure A File	
	return NULL; }	// If Not Return N	
	<pre>File=fopen(Filename, "r");</pre>	// Check To See If The File	
	if (File) {	// Does The File Exist?	
	<pre>fclose(File); return auxDIBImageLoad(Filename); }</pre>	// Close The Handl(// Load The Bitmap And Retu	
	return NULL;	// If Load Failed 1	

```
}
int LoadGLTextures()
                                                                         // Load Bitmaps And
{
         int Status=FALSE;
                                                                // Status Indicator
         AUX_RGBImageRec *TextureImage[1];
                                                                // Create Storage Space For
         memset(TextureImage,0,sizeof(void *)*1);
                                                               // Set The Pointer To NULL
         if (TextureImage[0]=LoadBMP("Data/Lights.bmp"))
                                                                         // Load The Bitmap
         {
                  Status=TRUE;
                                                                         // Set The Status :
                  glGenTextures(1, &texture[0]);
                                                                         // Create The Text
                  // Build Linear Mipmapped Texture
                  glBindTexture(GL_TEXTURE_2D, texture[0]);
                  gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAR)
```

The next four lines of code will automatically generate texture coordinates for any object we draw to the screen. The glTexGen command is extremely powerful, and complex, and to get into all the math involved would be a tutorial on it's own. All you need to know is that GL_S and GL_T are texture coordinates. By default they are set up to take the current x location on the screen and the current y location on the screen and come up with a texture vertex. You'll notice the objects are not textured on the z plane... just stripes appear. The front and back faces are textured though, and that's all that matters. X (GL_S) will cover mapping the texture left to right, and Y (GL_T) will cover mapping the texture up and down.

GL_TEXTURE_GEN_MODE lets us select the mode of texture mapping we want to use on the S and T texture coordinates. You have 3 choices:

GL_EYE_LINEAR - The texture is fixed to the screen. It never moves. The object is mapped with whatever section of the texture it is passing over.

GL_OBJECT_LINEAR - This is the mode we are using. The texture is fixed to the object moving around the screen.

GL_SPHERE_MAP - Everyones favorite. Creates a metalic reflective type object.

It's important to note that I'm leaving out alot of code. We should be setting the GL_OBJECT_PLANE as well, but by default it's set to the parameters we want. Buy a good book if you're interested in learning more, or check out the MSDN help CD / DVD.

```
// Texturing Contour Anchored To The Object
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
         // Texturing Contour Anchored To The Object
         glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
         glEnable(GL_TEXTURE_GEN_S);
                                                     // Auto Texture Generation
         glEnable(GL_TEXTURE_GEN_T);
                                                      // Auto Texture Generation
}
if (TextureImage[0])
                                                               // If Texture Exist
{
         if (TextureImage[0]->data)
                                                    // If Texture Image Exists
         {
                 free(TextureImage[0]->data);
                                                               // Free The Texture
         }
```

free(TextureImage[0]); // Free The Image :
}
return Status; // Return The Statu
}

,

}

There are a few new lines at the end of the InitGL() code. BuildFont() has been moved underneath our texture loading code. The line glEnable(GL_COLOR_MATERIAL) has been removed. If you plan to apply colors to the texture using glColor3f(r,g,b) add the line glEnable(GL_COLOR_MATERIAL) back into this section of code.

```
int InitGL(GLvoid)
                                                                // All Setup For OpenGL Goes
ł
         if (!LoadGLTextures())
                                                                         // Jump To Texture
         {
                  return FALSE;
                                                                         // If Texture Didn
         }
         BuildFont();
                                                                         // Build The Font
         glShadeModel(GL_SMOOTH);
                                                                // Enable Smooth Shading
         glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
                                                                         // Black Background
         glClearDepth(1.0f);
                                                                         // Depth Buffer Set
         glEnable(GL_DEPTH_TEST);
                                                                // Enables Depth Testing
         glDepthFunc(GL_LEQUAL);
                                                                         // The Type Of Dept
         glEnable(GL_LIGHT0);
                                                                         // Quick And Dirty
         glEnable(GL_LIGHTING);
                                                                         // Enable Lighting
         glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective (
```

Enable 2D Texture Mapping, and select texture one. This will map texture one onto any 3D object we draw to the screen. If you want more control, you can enable and disable texture mapping yourself.

```
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select The Texture
return TRUE; // Initialization I
```

The resize code hasn't changed, but our DrawGLScene code has.

```
int DrawGLScene(GLvoid) // Here's Where We
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The
    glLoadIdentity(); // Reset The View
```

Here's our first change. Instead of keeping the object in the middle of the screen, we're going to spin it around the screen using COS and SIN (no surprise). We'll translate 3 units into the screen (-3.0f). On the x axis, we'll swing from -1.1 at far left to +1.1 at the right. We'll be using the rot variable to control the left right swing. We'll swing from +0.8 at top to -0.8 at the bottom. We'll use the rot variable for this swinging motion as well. (might as well make good use of your variables).

```
// Position The Text
glTranslatef(1.1f*float(cos(rot/16.0f)),0.8f*float(sin(rot/20.0f)),-3.0f);
```

Now we do the normal rotations. This will cause the symbol to spin on the X, Y and Z axis.

glRotatef(rot,1.0f,0.0f,0.0f);

 glRotatef(rot*1.2f,0.0f,1.0f,0.0f);
 // Rotate On The Y Axis

 glRotatef(rot*1.4f,0.0f,0.0f,1.0f);
 // Rotate On The Z Axis

// Rotate On The X

We translate a little to the left, down, and towards the viewer to center the symbol on each axis. Otherwise when it spins it doesn't look like it's spinning around it's own center. -0.35 is just a number that worked. I had to play around with numbers for a bit because I'm not sure how wide the font is, could vary with each font. Why the fonts aren't built around a central point I'm not sure.

```
glTranslatef(-0.35f,-0.35f,0.1f);
```

// Center On X, Y, Z Axis

Finally we draw our skull and crossbones symbol then increase the rot variable so our symbol spins and moves around the screen. If you can't figure out how I get a skull and crossbones from the letter 'N', do this: Run Microsoft Word or Wordpad. Go to the fonts drop down menu. Select the Wingdings font. Type and uppercase 'N'. A skull and crossbones appears.

```
glPrint("N");
rot+=0.1f;
return TRUE;
```

// Draw A Skull And // Increase The Rot // Keep Going

}

}

The last thing to do is add KillFont() to the end of KillGLWindow() just like I'm showing below. It's important to add this line. It cleans things up before we exit our program.

```
if (!UnregisterClass("OpenGL",hInstance))
                                                      // Are We Able To Unregister
{
         MessageBox(NULL, "Could Not Unregister Class.", "SHUTDOWN ERROR", MB_OK | MB_
         hInstance=NULL;
                                                                // Set hInstance To
}
                                                                // Destroy The Foni
KillFont();
```

Even though I never went into extreme detail, you should have a pretty good understanding on how to make OpenGL generate texture coordinates for you. You should have no problems mapping textures to fonts of your own, or even other objects for that matter. And by changing just two lines of code, you can enable sphere mapping, which is a really cool effect.

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Visual Fortran Code For This Lesson. (Conversion by Jean-Philippe Perois)

Back To NeHe Productions!

Lesson 16

This tutorial brought to you by Chris Aliotta...

So you want to add fog to your OpenGL program? Well in this tutorial I will show you how to do exactly that. This is my first time writing a tutorial, and I am still relatively new to OpenGL/C++ programming, so please, if you find anything that's wrong let me know and don't jump all over me. This code is based on the code from lesson 7.

Data Setup:

We'll start by setting up all our variables needed to hold the information for fog. The variable fogMode will be used to hold three types of fog: GL_EXP, GL_EXP2, and GL_LINEAR. I will explain the differences between these three later on. The variables will start at the beginning of the code after the line GLuint texture[3]. The variable fogfilter will be used to keep track of which fog type we will be using. The variable fogColor will hold the color we wish the fog to be. I have also added the boolean variable gp at the top of the code so we can tell if the 'g' key is being pressed later on in this tutorial.

bool gp; GLuint filter; GLuint fogMode[]= { GL_EXP, GL_EXP2, GL_LINEAR }; // Storage For Three Types Of Fog GLuint fogfilter= 0; GLfloat fogColor[4]= {0.5f, 0.5f, 0.5f, 1.0f};

// G Pressed? (New) // Which Filter To Use // Which Fog To Use // Fog Color

DrawGLScene Setup

Now that we have established our variables we will move down to InitGL. The glClearColor() line has been modified to clear the screen to the same same color as the fog for a better effect. There isn't much code involved to make fog work. In all you will find this to be very simplistic.

glClearColor(0.5f,0.5f,0.5f,1.0f);

```
glFogi(GL_FOG_MODE, fogMode[fogfilter]);
glFoqfv(GL_FOG_COLOR, foqColor);
glFoqf(GL_FOG_DENSITY, 0.35f);
glHint(GL_FOG_HINT, GL_DONT_CARE);
glFogf(GL_FOG_START, 1.0f);
glFogf(GL_FOG_END, 5.0f);
glEnable(GL_FOG);
```

// We'll Clear To The Color Of The Fc

- // Fog Mode
- // Set Fog Color
 - // How Dense Will The Fog Be
- // Fog Hint Value
- // Fog Start Depth
- // Fog End Depth
- // Enables GL_FOG

Lets pick apart the first three lines of this code. The first line glEnable(GL_FOG); is pretty much self explanatory. It basically initializes the fog.

The second line, glFogi(GL_FOG_MODE, **fogMode**[**fogfilter**]); establishes the fog filter mode. Now earlier we declared the array **fogMode**. It held GL_EXP, GL_EXP2, and GL_LINEAR. Here is when these variables come into play. Let me explain each one:

- **GL_EXP** Basic rendered fog which fogs out all of the screen. It doesn't give much of a fog effect, but gets the job done on older PC's.
- **GL_EXP2** Is the next step up from GL_EXP. This will fog out all of the screen, however it will give more depth to the scene.
- **GL_LINEAR** This is the best fog rendering mode. Objects fade in and out of the fog much better.

The third line, glFogfv(GL_FOG_COLOR, **fogcolor**); sets the color of the fog. Earlier we had set this to (0.5f,0.5f,0.5f,1.0f) using the variable **fogcolor**, giving us a nice grey color.

Next lets look at the last four lines of this code. The line glFogf(GL_FOG_DENSITY, 0.35f); establishes how dense the fog will be. Increase the number and the fog becomes more dense, decrease it and it becomes less dense.

The line glHint (GL_FOG_HINT, GL_DONT_CARE); establishes the hint. I used GL_DONT_CARE, because I didn't care about the hint value. However here is an explanation of the different values for this option, provided by Eric Desrosiers:

Eric Desrosiers Adds: Little explanation of glHint(GL_FOG_HINT, hintval);

hintval can be : GL_DONT_CARE, GL_NICEST or GL_FASTEST

gl_dont_care - Lets opengl choose the kind of fog (per vertex of per pixel) and an unknown formula. gl_nicest - Makes the fog per pixel (look good) glfastest - Makes the fog per vertex (faster, but less nice)

The next line glFogf(GL_FOG_START, 1.0f); will establish how close to the screen the fog should start. You can change the number to whatever you want depending on where you want the fog to start. The next line is similar, glFogf(GL_FOG_END, 5.0f);. This tells the OpenGL program how far into the screen the fog should go.

Keypress Events

Now that we've setup the fog drawing code we will add the keyboard commands to cycle through the different fog modes. This code goes down at the end of the program with all the other key handling code.

```
if (keys['G'] && !qp)
                                                                 // Is The G Key Being Presse
{
         qp=TRUE;
                                                        // qp Is Set To TRUE
         fogfilter+=1;
                                                                // Increase fogfilter By On€
         if (fogfilter>2)
                                                        // Is fogfilter Greater Than 2?
         {
                  fogfilter=0;
                                                                 // If So, Set fogfilter To 2
         }
         glFogi (GL_FOG_MODE, fogMode[fogfilter]); // Fog Mode
if (!keys['G'])
                                                                 // Has The G Key Been Releas
{
         qp=FALSE;
                                                        // If So, qp Is Set To FALSE
```

}

That's it! We are done! You now have fog in your OpenGL program. I'd have to say that was pretty painless. If you have any questions or comments feel free to contact me at chris@incinerated.com. Also please stop by my website: http://www.incinerated.com/ and http://www.incinerated.com/precursor.

Christopher Aliotta (Precursor)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)
- * DOWNLOAD Java Code For This Lesson. (Conversion by Darren Hodges)

Back To NeHe Productions!

Lesson 17

This tutorial brought to you by NeHe & Giuseppe D'Agata...

I know everyones probably sick of fonts. The text tutorials I've done so far not only display text, they display 3D text, texture mapped text, and can handle variables. But what happens if you're porting your project to a machine that doesn't support Bitmap or Outline fonts?

Thanks to Giuseppe D'Agata we have yet another font tutorial. What could possibly be left you ask!? If you remember in the first Font tutorial I mentioned using textures to draw letters to the screen. Usually when you use textures to draw text to the screen you load up your favorite art program, select a font, then type the letters or phase you want to display. You then save the bitmap and load it into your program as a texture. Not very efficient for a program that require alot of text, or text that continually changes!

This program uses just ONE texture to display any of 256 different characters on the screen. Keep in mind your average character is just 16 pixels wide and roughly 16 pixels tall. If you take your standard 256x256 texture it's easy to see that you can fit 16 letters across, and you can have a total of 16 rows up and down. If you need a more detailed explanation: The texture is 256 pixels wide, a character is 16 pixels wide. 256 divided by 16 is 16 :)

So... Lets create a 2D textured font demo! This program expands on the code from lesson 1. In the first section of the program, we include the math and stdio libraries. We need the math library to move our letters around the screen using SIN and COS, and we need the stdio library to make sure the bitmaps we want to use actually exist before we try to make textures out of them.

```
#include <windows.h> // Header File For Windows
#include <math.h> // Header File For Windows Math Library (ADD)
#include <stdio.h> // Header File For Standard Input/Output (ADD)
#include <gl\gl.h> // Header File For The OpenGL32 Library
#include <gl\glu.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Holds Our Window Handle
#INSTANCEhInstance; // Holds The Instance Of The Application
#include Active=TRUE; // Window Active Flag Set To TRUE By Default
#include Sol fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
```

We're going to add a variable called **base** to point us to our display lists. We'll also add **texture[2]** to hold the two textures we're going to create. Texture 1 will be the font texture, and texture 2 will be a bump texture used to create our simple 3D object.

We add the variable **loop** which we will use to execute loops. Finally we add **cnt1** and **cnt2** which we will use to move the text around the screen and to spin our simple 3D object.

GLuint GLuint	<pre>base; texture[2];</pre>	// Base Display List For The Font // Storage For Our Font Texture	
GLuint	loop;	// Generic Loop Variable	
GLfloat	cnt1;	// 1st Counter Used To Move Text & For Color	ing
GLfloat	cnt2;	// 2nd Counter Used To Move Text & For Color	ing
LRESULT	CALLBACK WndProc(HWND, UIN	T, WPARAM, LPARAM);	// Declara

Now for the texture loading code. It's exactly the same as it was in the previous texture mapping tutorials.

```
AUX_RGBImageRec *LoadBMP(char *Filename)
                                                                                      // Loads 1
{
         FILE *File=NULL;
                                                                                      // File Ha
         if (!Filename)
         {
                  return NULL;
         }
         File=fopen(Filename,"r");
                                                                                      // Check '
         if (File)
                                                                                      // Does T
         {
                   fclose(File);
                                                                                      // Load Tl
                  return auxDIBImageLoad(Filename);
         }
         return NULL;
}
```

The follwing code has also changed very little from the code used in previous tutorials. If you're not sure what each of the following lines do, go back and review.

Note that **TextureImage[]** is going to hold 2 rgb image records. It's very important to double check code that deals with loading or storing our textures. One wrong number could result in a memory leak or crash!

```
int LoadGLTextures()
{
    int Status=FALSE;
    AUX_RGBImageRec *TextureImage[2];
```

// Status
// Create

The next line is the most important line to watch. If you were to replace the 2 with any other number, major problems will happen. Double check! This number should match the number you used when you set up **TextureImages[].**

The two textures we're going to load are font.bmp (our font), and bumps.bmp. The second texture can be replaced with any texture you want. I wasn't feeling very creative, so the texture I decided to use may be a little drab.

<pre>memset(TextureImage,0,sizeof(void *)*2);</pre>	// Set Th
if ((TextureImage[0]=LoadBMP("Data/Font.bmp")) &&	// Load T
(TextureImage[1]=LoadBMP("Data/Bumps.bmp")))	// Load T

Status=TRUE;

{

Another important line to double check. I can't begin to tell you how many emails I've received from people asking *"why am I only seeing one texture, or why are my textures all white!?!"*. Usually this line is the problem. If you were to replace the 2 with a 1, only one texture would be created and the second texture would appear all white. If you replaced the 2 with a 3 you're program may crash!

You should only have to call glGenTextures() once. After glGenTextures() you should generate all your textures. I've seen people put a glGenTextures() line before each texture they create. Usually they causes the new texture to overwrite any textures you've already created. It's a good idea to decide how many textures you need to build, call glGenTextures() once, and then build all the textures. It's not wise to put glGenTextures() inside a loop unless you have a reason to.

The following lines of code check to see if the bitmap data we loaded to build our textures is using up ram. If it is, the ram is freed. Notice we check and free both rgb image records. If we used 3 different images to build our textures, we'd check and free 3 rgb image records.

Now we're going to build our actual font. I'll go through this section of code in some detail. It's not really that complex, but there's a bit of math to understand, and I know math isn't something everyone enjoys.

GLvoid BuildFont(GLvoid)
{

}

// Build (

The following two variable will be used to hold the position of each letter inside the font texture. **cx** will hold the position from left to right inside the texture, and **cy** will hold the position up and down.

float cx; float cy;

Next we tell OpenGL we want to build 256 display lists. The variable **base** will point to the location of the first display list. The second display list will be **base+**1, the third will be **base+**2, etc.

The second line of code below selects our font texture (texture[0]).

```
base=glGenLists(256);
glBindTexture(GL_TEXTURE_2D, texture[0]);
```

// Select

Now we start our loop. The loop will build all 256 characters, storing each character in it's own display lists.

for (loop=0; loop<256; loop++)
{</pre>

The first line below may look a little puzzling. The % symbol means the remainder after **loop** is divided by 16. **cx** will move us through the font texture from left to right. You'll notice later in the code we subtract **cy** from 1 to move us from top to bottom instead of bottom to top. The % symbol is fairly hard to explain but I will make an attempt.

All we are really concerned about is (**loop**%16) the /16.0f just converts the results into texture coordinates. So if **loop** was equal to 16... **cx** would equal the remained of 16/16 which would be 0. but **cy** would equal 16/16 which is 1. So we'd move down the height of one character, and we wouldn't move to the right at all. Now if **loop** was equal to 17, **cx** would be equal to 17/16 which would be 1.0625. The remainder .0625 is also equal to 1/16th. Meaning we'd move 1 character to the right. **cy** would still be equal to 1 because we are only concerned with the number to the left of the decimal. 18/16 would gives us 2 over 16 moving us 2 characters to the right, and still one character down. If loop was 32, **cx** would once again equal 0, because there is no remained when you divide 32 by 16, but **cy** would equal 2. Because the number to the left of the decimal would now be 2, moving us down 2 characters from the top of our font texture. Does that make sense?

cx=float(loop%16)/16.0f; cy=float(loop/16)/16.0f; // X Posi // Y Posi

Whew :) Ok. So now we build our 2D font by selecting an individual character from our font texture depending on the value of **cx** and **cy**. In the line below we add **loop** to the value of **base** if we didn't, every letter would be built in the first display list. We definitely don't want that to happen so by adding loop to base, each character we create is stored in the next available display list.

// Start 1

Now that we've selected the display list we want to build, we create our character. This is done by drawing a quad, and then texturing it with just a single character from the font texture.

glBegin(GL_QUADS);

// Use A (

cx and **cy** should be holding a very tiny floating point value from 0.0f to 1.0f. If both **cx** and **cy** were equal to 0 the first line of code below would actually be: glTexCoord2f(0.0f,1-0.0f-0.0625f). Remember that 0.0625 is exactly 1/16th of our texture, or the width / height of one character. The texture coordinate below would be the bottom left point of our texture.

Notice we are using glVertex2i(x,y) instead of glVertex3f(x,y,z). Our font is a 2D font, so we don't need the z value. Because we are using an Ortho screen, we don't have to translate into the screen. All you have to do to draw to an Ortho screen is specify an x and y coordinate. Because our screen is in pixels from 0 to 639 and 0 to 479, we don't have to use floating point or negative values either :)

The way we set up our Ortho screen, (0,0) will be at the bottom left of our screen. (640,480) will be the top right of the screen. 0 is the left side of the screen on the x axis, 639 is the right side of the screen on the x axis. 0 is the bottom of the screen on the y axis and 479 is the top of the screen on the y axis. Basically we've gotten rid of negative coordinates. This is also handy for people that don't care about perspective and prefer to work with pixels rather than units :)

glTexCoord2f(cx,1-cy-0.0625f);
glVertex2i(0,0);

// Vertex

The next texture coordinate is now 1/16th to the right of the last texture coordinate (exactly one character wide). So this would be the bottom right texture point.

glTexCoord2f(cx+0.0625f,1-cy-0.0625f); glVertex2i(16,0);

// Vertex

The third texture coordinate stays at the far right of our character, but moves up 1/16th of our texture (exactly the height of one character). This will be the top right point of an individual character.

glTexCoord2f(cx+0.0625f,1-cy); glVertex2i(16,16); // Vertex

Finally we move left to set our last texture coordinate at the top left of our character.

glTexCoord2f(cx,1-cy); glVertex2i(0,16); // Vertex glEnd(); // Done B⁻ Finally, we translate 10 pixels to the right, placing us to the right of our texture. If we didn't translate, the letters would all be drawn on top of eachother. Because our font is so narrow, we don't want to move 16 pixels to the right. If we did, there would be big spaces between each letter. Moving by just 10 pixels eliminates the spaces.

```
glTranslated(10,0,0);
glEndList();
}
```

The following section of code is the same code we used in our other font tutorials to free the display list before our program quits. All 256 display lists starting at **base** will be deleted. (good thing to do!).

```
GLvoid KillFont(GLvoid)
{
    glDeleteLists(base,256);
}
```

```
// Delete
```

The next section of code is where all of our drawing is done. Everything is fairly new so I'll try to explain each line in great detail. Just a small note: Alot can be added to this code, such as variable support, character sizing, spacing, and alot of checking to restore things to how they were before we decided to print.

glPrint() takes three parameters. The first is the **x** position on the screen (the position from left to right). Next is the **y** position on the screen (up and down... 0 at the bottom, bigger numbers at the top). Then we have our actual **string** (the text we want to print), and finally a variable called **set**. If you have a look at the bitmap that Giuseppe D'Agata has made, you'll notice there are two different character sets. The first character set is normal, and the second character set is italicized. If **set** is 0, the first character set is selected.

```
GLvoid glPrint(GLint x, GLint y, char *string, int set) {
```

The first thing we do is make sure that **set** is either 0 or 1. If **set** is greater than 1, we'll make it equal to 1.

```
if (set>1)
{
    set=1;
}
```

Now we select our Font texture. We do this just in case a different texture was selected before we decided to print something to the screen.

glBindTexture(GL_TEXTURE_2D, texture[0]);

Now we disable depth testing. The reason I do this is so that blending works nicely. If you don't disable depth testing, the text may end up going behind something, or blending may not look right. If you have no plan to blend the text onto the screen (so that black spaces do not show up around our letters) you can leave depth testing on.

```
glDisable(GL_DEPTH_TEST);
```

The next few lines are VERY important! We select our Projection Matrix. Right after that, we use a command called glPushMatrix(). glPushMatrix stores the current matrix (projection). Kind of like the memory button on a calculator.

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
```

Now that our projection matrix has been stored, we reset the matrix and set up our Ortho screen. The first and third numbers (0) represent the bottom left of the screen. We could make the left side of the screen equal -640 if we want, but why would we work with negatives if we don't need to. The second and fourth numbers represent the top right of the screen. It's wise to set these values to match the resolution you are currently in.

```
glLoadIdentity();
glOrtho(0,640,0,480,-100,100);
```

Now we select our modelview matrix, and store it's current settings using glPushMatrix(). We then reset the modelview matrix so we can work with it using our Ortho view.

```
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
```

With our perspective settings saved, and our Ortho screen set up, we can now draw our text. We start by translating to the position on the screen that we want to draw our text at. We use glTranslated() instead of glTranslatef() because we are working with actual pixels, so floating point values are not important. After all, you can't have half a pixel :)

```
glTranslated(x,y,0);
```

The line below will select which font set we want to use. If we want to use the second font set we add 128 to the current base display list (128 is half of our 256 characters). By adding 128 we skip over the first 128 characters.

// Disabl

// Reset '

// Select
// Reset '

```
glListBase(base-32+(128*set));
```

Now all that's left for us to do is draw the letters to the screen. We do this exactly the same as we did in all the other font tutorials. We use glCallLists(). strlen(**sting**) is the length of our string (how many characters we want to draw), GL_BYTE means that each character is represented by a byte (a byte is any value from 0 to 255). Finally, **string** holds the actual text we want to print to the screen.

```
glCallLists(strlen(string),GL_BYTE,string);
```

All we have to do now is restore our perspective view. We select the projection matrix and use glPopMatrix() to recall the settings we previously stored with glPushMatrix(). It's important to restore things in the opposite order you stored them in.

```
glMatrixMode(GL_PROJECTION);
glPopMatrix();
```

Now we select the modelview matrix, and do the same thing. We use glPopMatrix() to restore our modelview matrix to what it was before we set up our Ortho display.

```
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
```

Finally, we enable depth testing. If you didn't disable depth testing in the code above, you don't need this line.

glEnable(GL_DEPTH_TEST);

}

Nothing has changed in ReSizeGLScene() so we'll skip right to InitGL().

```
int InitGL(GLvoid) {
```

We jump to our texture building code. If texture building fails for any reason, we return FALSE. This lets our program know that an error has occurred and the program gracefully shuts down.

// Select

// Write '

// Enable

// All Se

If there were no errors, we jump to our font building code. Not much can go wrong when building the font so we don't bother with error checking.

BuildFont();

Now we do our normal GL setup. We set the background clear color to black, the clear depth to 1.0. We choose a depth testing mode, along with a blending mode. We enable smooth shading, and finally we enable 2D texture mapping.

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClearDepth(1.0);
glDepthFunc(GL_LEQUAL);
glBlendFunc(GL_SRC_ALPHA,GL_ONE);
glShadeModel(GL_SMOOTH);
glEnable(GL_TEXTURE_2D);
return TRUE;
```

}

The section of code below will create our scene. We draw the 3D object first and the text last so that the text appears on top of the 3D object, instead of the 3D object covering up the text. The reason I decide to add a 3D object is to show that both perspective and ortho modes can be used at the same time.

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear '
    glLoadIdentity(); // Reset '
```

We select our bumps.bmp texture so that we can build our simple little 3D object. We move into the screen 5 units so that we can see the 3D object. We rotate on the z axis by 45 degrees. This will rotate our quad 45 degrees clockwise and makes our quad look more like a diamond than a square.

glBindTexture(GL_TEXTURE_2D, texture[1]);	// Select
glTranslatef(0.0f,0.0f,-5.0f);	
glRotatef(45.0f,0.0f,0.0f,1.0f);	// Rotate

After we have done the 45 degree rotation, we spin the object on both the x axis and y axis based on the variable **cnt1** times 30. This causes our object to spin around as if the diamond is spinning on a point.

glRotatef(cnt1*30.0f,1.0f,1.0f,0.0f);

// Enable

// Select

// Enable
// Enable

We disable blending (we want the 3D object to appear solid), and set the color to bright white. We then draw a single texture mapped quad.

glDisable(GL_BLEND);		
glColor3f(1.0f,1.0f,1.0f);		
glBegin(GL_QUADS);		
glTexCoord2d(0.0f,0.0f);	// First '	
glVertex2f(-1.0f, 1.0f);	// First '	
glTexCoord2d(1.0f,0.0f);	// Second	
glVertex2f(1.0f, 1.0f);	// Second	
glTexCoord2d(1.0f,1.0f);	// Third '	
glVertex2f(1.0f,-1.0f);	// Third '	
glTexCoord2d(0.0f,1.0f);	// Fourth	
glVertex2f(-1.0f,-1.0f);	// Fourth	
glEnd();	// Done D:	

Immediately after we've drawn the first quad, we rotate 90 degrees on both the x axis and y axis. We then draw another quad. The second quad cuts through the middle of the first quad, creating a nice looking shape.

glRotate	f(90.0f,1.0f,1.0f,0.0f);	//	Rotate
glBegin(GL_QUADS);		//	Draw O
	glTexCoord2d(0.0f,0.0f);	//	First '
	glVertex2f(-1.0f, 1.0f);	//	First '
	glTexCoord2d(1.0f,0.0f);	//	Second
	glVertex2f(1.0f, 1.0f);	//	Second
	glTexCoord2d(1.0f,1.0f);	//	Third '
	glVertex2f(1.0f,-1.0f);	//	Third '
	glTexCoord2d(0.0f,1.0f);	//	Fourth
	glVertex2f(-1.0f,-1.0f);	//	Fourth
glEnd();		//	Done D:

After both texture mapped quads have been drawn, we enable enable blending, and draw our text.

glEnable(GL_BLEND);
glLoadIdentity();

// Reset '

We use the same fancy coloring code from our other text tutorials. The color is changed gradually as the text moves across the screen.

// Pulsing Colors Based On Text Position
glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt:

Then we draw our text. We still use glPrint(). The first parameter is the x position. The second parameter is the y position. The third parameter ("NeHe") is the text to write to the screen, and the last parameter is the character set to use (0 - normal, 1 - italic).

As you can probably guess, we swing the text around the screen using COS and SIN, along with both counters **cnt1** and **cnt2**. If you don't understand what SIN and COS do, go back and read the previous text tutorials.

```
glPrint(int((280+250*cos(cnt1))),int(235+200*sin(cnt2)),"NeHe",0); // Print(
glColor3f(1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt2)),1.0f*float(cos(cnt2)),
glPrint(int((280+230*cos(cnt2))),int(235+200*sin(cnt1)),"OpenGL",1); // Print()
```

We set the color to a dark blue and write the author's name at the bottom of the screen. We then write his name to the screen again using bright white letters. The white letters are a little to the right of the blue letters. This creates a shadowed look. (if blending wasn't enabled the effect wouldn't work).

glColor3f(0.0f,0.0f,1.0f);	// Set Co
glPrint(int(240+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0);	// Draw T
glColor3f(1.0f,1.0f,1.0f);	// Set Co
glPrint(int(242+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0);	// Draw O

The last thing we do is increase both our counters at different rates. This causes the text to move, and the 3D object to spin.

```
cnt1+=0.01f;
cnt2+=0.0081f;
return TRUE;
```

}

The code in KillGLWindow(), CreateGLWindow() and WndProc() has not changed so we'll skip over it.

```
int WINAPI WinMain(
                         HINSTANCEhInstance,
                                                                              // Instan
                         HINSTANCEhPrevInstance,
                                                                              // Previo
                          LPSTR lpCmdLine,
                          int
                                           nCmdShow)
                                                                              // Window
{
        MSG
                         msq;
        BOOL done=FALSE;
        // Ask The User Which Screen Mode They Prefer
        if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreen"
        {
                 fullscreen=FALSE;
                                                                               // Window
        }
```

The title of our Window has changed.

}

```
// Create Our OpenGL Window
if (!CreateGLWindow("NeHe & Giuseppe D'Agata's 2D Font Tutorial",640,480,16,fullsc:
{
         return 0;
                                                                           // Quit I:
}
while(!done)
{
         if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
                                                                           // Is The:
         {
                  if (msg.message==WM_QUIT)
                                                                           // Have W
                   {
                            done=TRUE;
                  }
                  else
                   {
                            TranslateMessage(&msg);
                            DispatchMessage(&msg);
                   }
         }
         else
         {
                  // Draw The Scene. Watch For ESC Key And Quit Messages From Dra
                  if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active
                   {
                            done=TRUE;
                  }
                  else
                   {
                            SwapBuffers(hDC);
                                                                           // Swap B
                   }
         }
}
// Shutdown
```

The last thing to do is add KillFont() to the end of KillGLWindow() just like I'm showing below. It's important to add this line. It cleans things up before we exit our program.

```
if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To Unregister
{
    MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | MB_
    hInstance=NULL; // Set hInstance Tc
}
KillFont(); // Destroy The Font
```

I think I can officially say that my site now teaches every possible way to write text to the screen {grin}. All in all, I think this is a fairly good tutorial. The code can be used on any computer that can run OpenGL, it's easy to use, and writing text to the screen using this method requires very little processing power.

I'd like to thank Giuseppe D'Agata for the original version of this tutorial. I've modified it heavily, and converted it to the new base code, but without him sending me the code I probably wouldn't have written the tutorial. His version of the code had a few more options, such as spacing the characters, etc, but I make up for it with the extremely cool 3D object {grin}.

I hope everyone enjoys this tutorial. If you have questions, email Giuseppe D'Agata or myself.

Giuseppe D'Agata

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Jörgen Isaksson)

Back To NeHe Productions!

Lesson 18

Quadratics

Quadratics are a way of drawing complex objects that would usually take a few for loops and some background in trigonometry.

We'll be using the code from lesson seven. We will add 7 variables and modify the texture to add some variety :)

```
#include <windows.h>
                                                               // Header File For Windows
#include <stdio.h>
                                                      // Header File For Standard Input/Out
                                                      // Header File For The OpenGL32 Libra
#include <ql\ql.h>
                                                               // Header File For The GLu32
#include <gl\glu.h>
                                                               // Header File For The GLaux
#include <gl\glaux.h>
                 hDC=NULL;
                                                      // Private GDI Device Context
HDC
HGLRC
                 hRC=NULL;
                                                      // Permanent Rendering Context
HWND
                 hWnd=NULL;
                                                               // Holds Our Window Handle
HINSTANCEhInstance;
                                                      // Holds The Instance Of The Applicat
      keys[256];
bool
                                                               // Array Used For The Keyboa
bool
        active=TRUE;
                                                               // Window Active Flag Set To
bool
        fullscreen=TRUE;
                                                      // Fullscreen Flag Set To Fullscreen
bool
        light;
                                                               // Lighting ON/OFF
        lp;
bool
                                                               // L Pressed?
bool
       fp;
                                                               // F Pressed?
bool sp;
                                                               // Spacebar Pressed?
       part1;
int
                                                               // Start Of Disc ( NEW )
       part2;
int
                                                               // End Of Disc
      p1=0;
int
                                                               // Increase 1
int
        p2=1;
                                                               // Increase 2
GLfloat xrot;
                                                               // X Rotation
GLfloat yrot;
                                                               // Y Rotation
GLfloat xspeed;
                                                               // X Rotation Speed
GLfloat yspeed;
                                                               // Y Rotation Speed
GLfloat z=-5.0f;
                                                      // Depth Into The Screen
GLUquadricObj *quadratic;
                                                      // Storage For Our Quadratic Objects
GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f }; // Ambient Light Values
GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f }; // Diffuse Light Values
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f }; // Light Position
GLuint
       filter;
                                                               // Which Filter To Use
GLuint
        texture[3];
                                                               // Storage for 3 textures
GLuint object=0;
                                                      // Which Object To Draw
                                                                                ( NEW )
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc
```

Okay now move down to InitGL(), We're going to add 3 lines of code here to initialize our quadratic. Add these 3 lines after you enable light1 but before you return true. The first line of code initializes the Quadratic and creates a pointer to where it will be held in memory. If it can't be created it returns 0. The second line of code creates smooth normals on the quadratic so lighting will look great. Other possible values are GLU_NONE, and GLU_FLAT. Last we enable texture mapping on our quadratic. Texture mapping is kind of awkward and never goes the way you planned as you can tell from the crate texture.

quadratic=gluNewQuadric(); gluQuadricNormals(quadratic, GLU_SMOOTH); gluQuadricTexture(quadratic, GL_TRUE); Now I decided to keep the cube in this tutorial so you can see how the textures are mapped onto the quadratic object. I decided to move the cube into its own function so when we write the draw function it will appear more clean. Everybody should recognize this code. =P

```
GLvoid glDrawCube()
                                                                  // Draw A Cube
                  glBegin(GL_QUADS);
                                                        // Start Drawing Quads
                   // Front Face
                  glNormal3f( 0.0f, 0.0f, 1.0f);
                                                                  // Normal Facing Forward
                  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom
                  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
                                                                                    // Bottom
                  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
                                                                                    // Top Rig
                  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
                                                                                    // Top Le:
                   // Back Face
                  glNormal3f( 0.0f, 0.0f, -1.0f);
                                                                 // Normal Facing Away
                  glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom
                  glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
                                                                                    // Top Rig
                  glTexCoord21(1.01, 1.01); glVertex31(-1.01, 1.01, -1.01);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex2f( 1.0f, 1.0f, -1.0f);
                                                                                    // Top Le:
                  glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
                                                                                    // Bottom
                   // Top Face
                  glNormal3f( 0.0f, 1.0f, 0.0f);
                                                                 // Normal Facing Up
                  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Le
                                                                                    // Bottom
                  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
                  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
                                                                                    // Bottom
                  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
                                                                                    // Top Rig
                   // Bottom Face
                  glNormal3f( 0.0f,-1.0f, 0.0f);
                                                                 // Normal Facing Down
                  glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
                                                                                    // Top Rig
                  glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
                                                                                    // Top Le:
                  glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
                                                                                    // Bottom
                  glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
                                                                                    // Bottom
                   // Right face
                  glNormal3f( 1.0f, 0.0f, 0.0f);
                                                                 // Normal Facing Right
                  glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Bottom
                  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
                                                                                    // Top Rig
                  glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
                                                                                    // Top Le:
                  glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
                                                                                    // Bottom
                   // Left Face
                  glNormal3f(-1.0f, 0.0f, 0.0f);
                                                                 // Normal Facing Left
                  glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom
                  glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
                                                                                    // Bottom
                  glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
                                                                                    // Top Ri
                  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
                                                                                    // Top Le:
         glEnd();
                                                        // Done Drawing Quads
```

}

Next is the DrawGLScene function, here I just wrote a simple if statement to draw the different objects. Also I used a static variable (a local variable that keeps its value everytime it is called) for a cool effect when drawing the partial disk. I'm going to rewrite the whole DrawGLScene function for clarity.

You'll notice that when I talk about the parameters being used I ignore the actual first parameter (quadratic). This parameter is used for all the objects we draw aside from the cube, so I ignore it when I talk about the parameters.

```
int DrawGLScene(GLvoid)
                                                                         // Here's Where We
        glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT); // Clear The Screen And The
        glLoadIdentity();
                                                                // Reset The View
        glTranslatef(0.0f,0.0f,z);
                                                                // Translate Into The Screer
         glRotatef(xrot,1.0f,0.0f,0.0f);
                                                                         // Rotate On The X
        glRotatef(yrot,0.0f,1.0f,0.0f);
                                                                         // Rotate On The Y
         glBindTexture(GL_TEXTURE_2D, texture[filter]);
                                                                         // Select A Filter
         // This Section Of Code Is New ( NEW )
         switch(object)
                                                                         // Check object To
         {
         case 0:
                                                                         // Drawing Object 1
                                                                         // Draw Our Cube
                  glDrawCube();
                  break;
                                                                         // Done
```

The second object we create is going to be a Cylinder. The first parameter (1.0f) is the radius of the cylinder at base (bottom). The second parameter (1.0f) is the radius of the cylinder at the top. The third parameter (3.0f) is the height of the cylinder (how long it is). The fouth parameter (32) is how many subdivisions there are "around" the Z axis, and finally, the fifth parameter (32) is the amount of subdivisions "along" the Z axis. The more subdivisions there are the more detailed the object is. By increase the amount of subdivisions you add more polygons to the object. So you end up sacrificing speed for quality. Most of the time it's easy to find a happy medium.

```
case 1: // Drawing Object :
    glTranslatef(0.0f,0.0f,-1.5f); // Center The Cylin
    gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // Draw Our Cylinder
    break; // Done
```

The third object we create will be a CD shaped disc. The first parameter (0.5f) is the inner radius of the disk. This value can be zero, meaning there will be no hole in the middle. The larger the inner radius is, the bigger the hole in the middle of the disc will be. The second parameter (1.5f) is the outer radius. This value should be larger than the inner radius. If you make this value a little bit larger than the inner radius you will end up with a thing ring. If you make this value alot larger than the inner radius you will end up with a thick ring. The third parameter (32) is the number of slices that make up the disc. Think of slices like the slices in a pizza. The more slices you have, the smoother the outer edge of the disc will be. Finally the fourth parameter (32) is the number of rings that make up the disc. The rings are are similar to the tracks on a record. Circles inside circles. These ring subdivide the disc from the inner radius to the outer radius, adding more detail. Again, the more subdivisions there are, the slow it will run.

case 2: gluDisk(quadratic,0.5f,1.5f,32,32); // Draw A Disc (CD Shape) break;

// Drawing Object 1 // Done

Our fourth object is an object that I know many of you have been dying to figure out. The Sphere! This one is quite simple. The first parameter is the radius of the sphere. In case you're not familiar with radius/diameter, etc, the radius is the distance from the center of the object to the outside of the object. In this case our radius is 1.3f. Next we have our subdivision "around" the Z axis (32), and our subdivision "along" the Z axis (32). The more subdivisions you have the smoother the sphere will look. Spheres usually require quite a few subdivisions to make them look smooth.

case 3:		// Drawing Object
	gluSphere(quadratic,1.3f,32,32);	// Draw A Sphere
	break;	// Done

Our fifth object is created using the same command that we used to create a Cylinder. If you remember, when we were creating the Cylinder the first two parameters controlled the radius of the cylinder at the bottom and the top. To make a cone it makes sense that all we'd have to do is make the radius at one end Zero. This will create a point at one end. So in the code below, we make the radius at the top of the cylinder equal zero. This creates our point, which also creates our cone.

```
case 4:
                                                                // Drawing Object !
         glTranslatef(0.0f,0.0f,-1.5f);
                                                                // Center The Cone
         gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32); // A Cone With A Bottom Radi
        break;
                                                                // Done
```

Our sixth object is created with gluPartialDisc. The object we create using this command will look exactly like the disc we created above, but with the command gluPartialDisk there are two new parameters. The fifth parameter (part1) is the start angle we want to start drawing the disc at. The sixth parameter is the sweep angle. The sweep angle is the distance we travel from the current angle. We'll increase the sweep angle, which causes the disc to be slowly drawn to the screen in a clockwise direction. Once our sweep hits 360 degrees we start to increase the start angle. the makes it appear as if the disc is being erased, then we start all over again!

```
case 5:
                                                                  // Drawing Object (
                                                                  // Increase Start i
         part1+=p1;
                                                                  // Increase Sweep i
         part2+=p2;
         if(part1>359)
                                                                  // 360 Degrees
         {
                  p1=0;
                                                                 // Stop Increasing
                                                        // Set Start Angle To Zero
                  part1=0;
                                                                 // Start Increasing
                  p2=1;
                  part2=0;
                                                        // Start Sweep Angle At Zerc
         if(part2>359)
                                                                  // 360 Degrees
         {
                  p1=1;
                                                                  // Start Increasing
                  p2=0;
                                                                  // Stop Increasing
         }
         gluPartialDisk(quadratic,0.5f,1.5f,32,32,part1,part2-part1);
                                                                           // A Disk
         break;
                                                                  // Done
```

Jeff Molofee's OpenGL Windows Tutorial #18 (By GB Schmick)

```
};
xrot+=xspeed;
yrot+=yspeed;
return TRUE;
```

```
// Increase Rotati(
// Increase Rotati(
// Keep Going
```

}

Now for the final part, they key input. Just add this where we check the rest of key input.

```
if (keys[' '] && !sp) // Is Spacebar Bein
{
    sp=TRUE; // If So, Set sp To TRUE
    object++; // Cycle Through The Objects
    if(object>5) // Is object Greate
        object=0;// If So, Set To Zero
}
if (!keys[' ']) // Has The Spaceban
{
    sp=FALSE; // If So, Set sp To FALSE
}
```

Thats all! Now you can draw quadratics in OpenGL. Some really impressive things can be done with morphing and quadratics. The animated disc is an example of simple morphing.

GB Schmick (**TipTup**)

Everyone if you have time go check out my website, TipTup.Com 2000.

* DOWNLOAD Visual C++ Code For This Lesson.

- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)

Back To NeHe Productions!

Lesson 19

Welcome to Tutorial 19. You've learned alot, and now you want to play. I will introduce one new command in this tutorial... The triangle strip. It's very easy to use, and can help speed up your programs when drawing alot of triangles.

In this tutorial I will teach you how to make a semi-complex Particle Engine. Once you understand how particle engines work, creating effects such as fire, smoke, water fountains and more will be a piece of cake!

I have to warn you however! Until today I had never written a particle engine. I had this idea that the 'famous' particle engine was a very complex piece of code. I've made attempts in the past, but usually gave up after I realized I couldn't control all the points without going crazy.

You might not believe me when I tell you this, but this tutorial was written 100% from scratch. I borrowed no ones ideas, and I had no technical information sitting in front of me. I started thinking about particles, and all of a sudden my head filled with ideas (brain turning on?). Instead of thinking about each particle as a pixel that had to go from point 'A' to point 'B', and do this or that, I decided it would be better to think of each particle as an individual object responding to the environment around it. I gave each particle life, random aging, color, speed, gravitational influence and more.

Soon I had a finished project. I looked up at the clock and realized aliens had come to get me once again. Another 4 hours gone! I remember stopping now and then to drink coffee and blink, but 4 hours...?

So, although this program in my opinion looks great, and works exactly like I wanted it to, it may not be the proper way to make a particle engine. I don't care personally, as long as it works well, and I can use it in my projects! If you are the type of person that needs to know you're conforming, then spend hours browsing the net looking for information. Just be warned. The few code snippits you do find may appear cryptic :)

This tutorial uses the base code from lesson 1. There is alot of new code however, so I'll rewrite any section of code that contains changes (makes it easier to understand).

Using the code from lesson 1, we'll add 5 new lines of code at the top of our program. The first line (stdio.h) allows us to read data from files. It's the same line we've added to previous tutorials the use texture mapping. The second line defines how many particles were going to create and display on the screen. Define just tells our program that **MAX_PARTICLES** will equal whatever value we specify. In this case 1000. The third line will be used to toggle 'rainbow mode' off and on. We'll set it to on by default. **sp** and **rp** are variables we'll use to prevent the spacebar or return key from rapidly repeating when held down.

#include	<windows.h></windows.h>	// Header File For Windows
#include	<stdio.h></stdio.h>	// Header File For Standard Input/Output (ADD)
#include	<gl\gl.h></gl\gl.h>	// Header File For The OpenGL32 Library
#include	<gl\glu.h></gl\glu.h>	// Header File For The GLu32 Library
#include	<gl\glaux.h></gl\glaux.h>	// Header File For The GLaux Library
#define	MAX_PARTICLES 1000	// Number Of Particles To Create (NEW)
HDC	hDC=NULL;	// Private GDI Device Context

Jeff Molofee's OpenGL Windows Tutorial #19

HGLRC	hRC=NULL;	// Permanent Rendering Context
HWND	hWnd=NULL;	// Holds Our Window Handle
HINSTANC	EhInstance;	// Holds The Instance Of The Application
bool	keys[256];	// Array Used For The Keyboard Routine
bool	active=TRUE;	<pre>// Window Active Flag Set To TRUE By Default</pre>
bool	fullscreen=TRUE;	// Fullscreen Flag Set To Fullscreen Mode By Default
bool	rainbow=true;	// Rainbow Mode? (ADD)
bool	sp;	// Spacebar Pressed? (ADD)
bool	rp;	// Return Key Pressed? (ADD)

The next 4 lines are misc variables. The variable **slowdown** controls how fast the particles move. The higher the number, the slower they move. The lower the number, the faster they move. If the value is set to low, the particles will move way too fast! The speed the particles travel at will affect how they move on the screen. Slow particles will not shoot out as far. Keep this in mind.

The variables **xspeed** and **yspeed** allow us to control the direction of the tail. **xspeed** will be added to the current speed a particle is travelling on the x axis. If **xspeed** is a positive value our particle will be travelling more to the right. If **xspeed** is a negative value, our particle will travel more to the left. The higher the value, the more it travels in that direction. **yspeed** works the same way, but on the y axis. The reason I say 'MORE' in a specific direction is because other factors affect the direction our particle travels. **xspeed** and **yspeed** help to move the particle in the direction we want.

Finally we have the variable **zoom**. We use this variable to pan into and out of our scene. With particle engines, it's nice to see more of the screen at times, and cool to zoom in real close other times.

float	<pre>slowdown=2.0f;</pre>	// Slow Down Particles
float	xspeed;	// Base X Speed (To Allow Keyboard Direction C
float	yspeed;	// Base Y Speed (To Allow Keyboard Direction C
float	zoom=-40.0f;	// Used To Zoom Out

Now we set up a misc loop variable called **loop**. We'll use this to predefine the particles and to draw the particles to the screen. **col** will be use to keep track of what color to make the particles. **delay** will be used to cycle through the colors while in rainbow mode.

Finally, we set aside storage space for one texture (the particle texture). I decided to use a texture rather than OpenGL points for a few reasons. The most important reason is because points are not all that fast, and they look pretty blah. Secondly, textures are way more cool :) You can use a square particle, a tiny picture of your face, a picture of a star, etc. More control!

GLuint	loop;	// Misc Loop Variable
GLuint	col;	// Current Color Selection
GLuint	delay;	// Rainbow Effect Delay
GLuint	<pre>texture[1];</pre>	// Storage For Our Particle Texture

Ok, now for the fun stuff. The next section of code creates a structure describing a single particle. This is where we give the particle certain characteristics.

We start off with the boolean variable **active**. If this variable is TRUE, our particle is alive and kicking. If it's FALSE our particle is dead or we've turned it off! In this program I don't use **active**, but it's handy to include.

The variables **life** and **fade** control how long the particle is displayed, and how bright the particle is while it's alive. The variable **life** is gradually decreased by the value stored in fade. In this program that will cause some particles to burn longer than others.

typedef struct {		// Create A Structure For Pa
bool	active;	// Active (Yes/No)
float	life;	// Particle Life
float	fade;	// Fade Speed

The variables \mathbf{r} , \mathbf{g} and \mathbf{b} hold the red intensity, green intensity and blue intensity of our particle. The closer \mathbf{r} is to 1.0f, the more red the particle will be. Making all 3 variables 1.0f will create a white particle.

float	r;	11	Red Value
float	g;	//	Green Value
float	b;	//	Blue Value

The variables \mathbf{x} , \mathbf{y} and \mathbf{z} control where the particle will be displayed on the screen. \mathbf{x} holds the location of our particle on the x axis. \mathbf{y} holds the location of our particle on the y axis, and finally \mathbf{z} holds the location of our particle on the z axis.

float	x;	// X Position
float	y;	// Y Position
float	zi	// Z Position

The next three variables are important. These three variables control how fast a particle is moving on specific axis, and what direction to move. If **xi** is a negative value our particle will move left. Positive it will move right. If **yi** is negative our particle will move down. Positive it will move up. Finally, if **zi** is negative the particle will move into the screen, and postive it will move towards the viewer.

float	xi;	// X Direction
float	yi;	// Y Direction
float	zi;	// Z Direction

Lastly, 3 more variables! Each of these variables can be thought of as gravity. If **xg** is a positive value, our particle will pull to the right. If it's negative our particle will be pulled to the left. So if our particle is moving left (negative) and we apply a positive gravity, the speed will eventually slow so much that our particle will start moving the opposite direction. **yg** pulls up or down and **zg** pulls towards or away from the viewer.

Jeff Molofee's OpenGL Windows Tutorial #19

float	xg;
float	yg;
float	zg;

particles is the name of our structure.

}
particles;

// Particles Structure

// X Gravity
// Y Gravity
// Z Gravity

Next we create an array called **particle**. This array will store **MAX_PARTICLES**. Translated into english we create storage for 1000 (**MAX_PARTICLES**) particles. This storage space will store the information for each individual particle.

```
particles particle[MAX_PARTICLES];
```

// Particle Array (Room For Particle

We cut back on the amount of code required for this program by storing our 12 different colors in a color array. For each color from 1 to 12 we store the red intensity, the green intensity, and finally the blue intensity. The color table below stores 12 different colors fading from red to violet.

Our bitmap loading code hasn't changed.

```
AUX_RGBImageRec *LoadBMP(char *Filename)
                                                     // Loads A Bitmap Image
{
         FILE *File=NULL;
                                                       // File Handle
         if (!Filename)
                                                                // Make Sure A Filename Was
         {
                                                                // If Not Return NULL
                  return NULL;
         }
         File=fopen(Filename,"r");
                                                       // Check To See If The File Exists
         if (File)
                                                       // Does The File Exist?
         {
                  fclose(File);
                                                                // Close The Handle
                  return auxDIBImageLoad(Filename);
                                                       // Load The Bitmap And Return A Point
         }
         return NULL;
                                                                // If Load Failed Return NUI
}
```

}

This is the section of code that loads the bitmap (calling the code above) and converts it into a textures. Status is used to keep track of whether or not the texture was loaded and created.

int LoadGLTextures()	// Load Bitmaps And
int Status=FALSE;	// Status Indicator
AUX_RGBImageRec *TextureImage[1];	// Create Storage Space For
<pre>memset(TextureImage,0,sizeof(void *)*1);</pre>	// Set The Pointer To NULL

Our texture loading code will load in our particle bitmap and convert it to a linear filtered texture.

```
if (TextureImage[0]=LoadBMP("Data/Particle.bmp")) // Load Particle Texture
{
         Status=TRUE;
                                                                // Set The Status :
         glGenTextures(1, &texture[0]);
                                                                // Create One Texti
         glBindTexture(GL_TEXTURE_2D, texture[0]);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
         glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]
}
if (TextureImage[0])
                                                                // If Texture Exist
{
         if (TextureImage[0]->data)
                                                     // If Texture Image Exists
         {
                  free(TextureImage[0]->data);
                                                                // Free The Texture
         }
         free(TextureImage[0]);
                                                                // Free The Image :
}
return Status;
                                                                // Return The Stati
```

The only change I made to the resize code was a deeper viewing distance. Instead of 100.0f, we can now view particles 200.0f units into the screen.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
                                                            // Resize And Initialize The
{
         if (height==0)
                                                                        // Prevent A Divide
         {
                 height=1;
                                                               // Making Height Equal One
         }
        glViewport(0, 0, width, height);
                                                               // Reset The Current Viewpon
        glMatrixMode(GL_PROJECTION);
                                                                        // Select The Proje
         glLoadIdentity();
                                                               // Reset The Projection Mati
         // Calculate The Aspect Ratio Of The Window
         gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,200.0f);
                                                                                 ( MODIFI
```

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

}

// Select The Modelview Mat
// Reset The Modelview Matri

// All Se

If you're using the lesson 1 code, replace it with the code below. I've added code to load in our texture and set up blending for our particles.

We enable smooth shading, clear our background to black, enable depth testing, blending and texture mapping. After enabling texture mapping we select our particle texture.

glShadeModel(GL_SMOOTH);	// Enable
glClearColor(0.0f,0.0f,0.0f,0.0f);	// Black i
glClearDepth(1.0f);	
glEnable(GL_DEPTH_TEST);	// Enable
glEnable(GL_BLEND);	
glBlendFunc(GL_SRC_ALPHA,GL_ONE);	// Type O
glBlendFunc(GL_SRC_ALPHA,GL_ONE); glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);	// Type O // Really
glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);	

The code below will initialize each of the particles. We start off by activating each particle. If a particle is not active, it won't appear on the screen, no matter how much life it has.

After we've made the particle active, we give it life. I doubt the way I apply life, and fade the particles is the best way, but once again, it works good! Full life is 1.0f. This also gives the particle full brightness.

We set how fast the particle fades out by giving **fade** a random value. The variable **life** will be reduced by fade each time the particle is drawn. The value we end up with will be a random value from 0 to 99. We then divide it by 1000 so that we get a very tiny floating point value. Finally we then add .003 to the final result so that the fade speed is never 0.

particle[loop].fade=float(rand()%100)/1000.0f+0.003f; // Random

Now that our particle is active, and we've given it life, it's time to give it some color. For the initial effect, we want each particle to be a different color. What I do is make each particle one of the 12 colors that we've built in our color table at the top of this program. The math is simple. We take our **loop** variable and add one to it to prevent a divide by zero error. Then we divide **loop** by the number of particles we plan to create divided by the number of colors in our table +1.

If loop is 0 the result would be 0+1/(1000/12)=0.012. Because the result is an integer value, that will be rounded down to 0 (our first color). If loop was 1000 (maximum amount of particles), the result would be 1000+1/(1000/12)=12.012. Rounded as an integer the result would be 12 which is our last color.

<pre>particle[loop].r=colors[(loop+1)/(MAX_PARTICLES/12)][0];</pre>	// Select
<pre>particle[loop].g=colors[(loop+1)/(MAX_PARTICLES/12)][1];</pre>	// Select
<pre>particle[loop].b=colors[(loop+1)/(MAX_PARTICLES/12)][2];</pre>	// Select

Now we'll set the direction that each particle moves, along with the speed. We're going to multiply the results by 10.0f to create a spectacular explosion when the program first starts.

We'll end up with either a positive or negative random value. This value will be used to move the particle in a random direction at a random speed.

<pre>particle[loop].xi=float((rand()%50)-26.0f)*10.0f;</pre>	// Random
<pre>particle[loop].yi=float((rand()%50)-25.0f)*10.0f;</pre>	// Random
<pre>particle[loop].zi=float((rand()%50)-25.0f)*10.0f;</pre>	// Random

Finally, we set the amount of gravity acting on each particle. Unlike regular gravity that just pulls things down, our gravity can pull up, down, left, right, forward or backward. To start out we want semi strong gravity pulling downwards. To do this we set xg to 0.0f. No pull left or right on the x plane. We set yg to -0.8f. This creates a semi-strong pull downwards. If the value was positive it would pull upwards. We don't want the particles pulling towards or away from us so we'll set zg to 0.0f.

```
particle[loop].xg=0.0f;
particle[loop].yg=-0.8f;
particle[loop].zg=0.0f;
}
return TRUE;
```

// Set Ve:

Now for the fun stuff. The next section of code is where we draw the particle, check for gravity, etc. It's important that you understand what's going on, so please read carefully :)

We reset the Modelview Matrix only once. We'll position the particles using the glVertex3f() command instead of using tranlations, that way we don't alter the modelview matrix while drawing our particles.

}

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();

// Clear : // Reset '

We start off by creating a loop. This loop will update each one of our particles.

```
for (loop=0;loop<MAX_PARTICLES;loop++)
{</pre>
```

First thing we do is check to see if the particle is active. If it's not active, it wont be updated. In this program they're all active, all the time. But in a program of your own, you may want to make certain particles inactive.

```
if (particle[loop].active) // If The
{
```

The next three variables **x**, **y** and **z** are temporary variables that we'll use to hold the particles x, y and z position. Notice we add **zoom** to the z position so that our scene is moved into the screen based on the value stored in zoom. **particle[loop].x** holds our x position for whatever particle we are drawing (particle **loop**). **particle[loop].y** holds our y position for our particle and **particle [loop].z** holds our z position.

float x=particle[loop].x;
float y=particle[loop].y;
float z=particle[loop].z+zoom;

// Grab 0 // Grab 0

Now that we have the particle position, we can color the particle. **particle[loop].r** holds the red intensity of our particle, **particle[loop].g** holds our green intensity, and **particle[loop].b** holds our blue intensity. Notice I use the particles life for the alpha value. As the particle dies, it becomes more and more transparent, until it eventually doesn't exist. That's why the particles life should never be more than 1.0f. If you need the particles to burn longer, try reducing the fade speed so that the particle doesn't fade out as fast.

// Draw The Particle Using Our RGB Values, Fade The Particle Bas
glColor4f(particle[loop].r,particle[loop].g,particle[loop].b,par

We have the particle position and the color is set. All that we have to do now is draw our particle. Instead of using a textured quad, I've decided to use a textured triangle strip to speed the program up a bit. Most 3D cards can draw triangles alot faster than they can draw quads. Some 3D cards will convert the quad to two triangles for you, but some don't. So we'll do the work ourselves. We start off by telling OpenGL we want to draw a triangle strip.

glBegin(GL_TRIANGLE_STRIP);

// Build (



Quoted directly from the red book: A triangle strip draws a series of triangles (three sided polygons) using vertices V_0 , V_1 , V_2 , then V_2 , V_1 , V_3 (note the order), then V_2 , V_3 , V_4 , and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. Preserving the orientation is important for some operations, such as culling. There must be at least 3 points for anything to be drawn.

So the first triangle is drawn using vertices 0, 1 and 2. If you look at the picture you'll see that vertex points 0, 1 and 2 do indeed make up the first triangle (top right, top left, bottom right). The second triangle is drawn using vertices 2, 1 and 3. Again, if you look at the picture, vertices 2, 1 and 3 create the second triangle (bottom right, top left, bottom left). Notice that both triangles are drawn with the same winding (counter-clockwise orientation). I've seen quite a few web sites that claim every second triangle is wound the opposite direction. This is not the case. OpenGL will rearrange the vertices to ensure that all of the triangles are wound the same way!

There are two good reasons to use triangle strips. First, after specifying the first three vertices for the initial triangle, you only need to specify a single point for each additional triangle. That point will be combined with 2 previous vertices to create a triangle. Secondly, by cutting back the amount of data needed to create a triangle your program will run quicker, and the amount of code or data required to draw an object is greatly reduced.

Note: The number of triangles you see on the screen will be the number of vertices you specify minus 2. In the code below we have 4 vertices and we see two triangles.

```
glTexCoord2d(1,1); glVertex3f(x+0.5f,y+0.5f,z); // Top
glTexCoord2d(0,1); glVertex3f(x-0.5f,y+0.5f,z); // Top
glTexCoord2d(1,0); glVertex3f(x+0.5f,y-0.5f,z); // Bott
glTexCoord2d(0,0); glVertex3f(x-0.5f,y-0.5f,z); // Bott
```

Finally we tell OpenGL that we are done drawing our triangle strip.

glEnd();

// Done B

Now we can move the particle. The math below may look strange, but once again, it's pretty simple. First we take the current particle x position. Then we add the x movement value to the particle divided by **slowdown** times 1000. So if our particle was in the center of the screen on the x axis (0), our movement variable (**xi**) for the x axis was +10 (moving us to the right) and **slowdown** was equal to 1, we would be moving to the right by 10/(1*1000), or 0.01f. If we increase the slowdown to 2 we'll only be moving at 0.005f. Hopefully that helps you understand how **slowdown** works.

That's also why multiplying the start values by 10.0f made the pixels move alot faster, creating an explosion.

We use the same formula for the y and z axis to move the particle around on the screen.

```
particle[loop].y+=particle[loop].yi/(slowdown*1000); // Move 0:
particle[loop].z+=particle[loop].zi/(slowdown*1000); // Move 0:
```

After we've calculated where to move the particle to next, we have to apply gravity or resistance. In the first line below, we do this by adding our resistance (xg) to the speed we are moving at (xi).

Lets say our moving speed was 10 and our resistance was 1. Each time our particle was drawn resistance would act on it. So the second time it was drawn, resistance would act, and our moving speed would drop from 10 to 9. This causes the particle to slow down a bit. The third time the particle is drawn, resistance would act again, and our moving speed would drop to 8. If the particle burns for more than 10 redraws, it will eventually end up moving the opposite direction because the moving speed would become a negative value.

The resistance is applied to the y and z moving speed the same way it's applied to the x moving speed.

particle[loop].xi+=particle[loop].xg; particle[loop].yi+=particle[loop].yg; particle[loop].zi+=particle[loop].zg;

The next line takes some life away from the particle. If we didn't do this, the particle would never burn out. We take the current life of the particle and subtract the fade value for that particle. Each particle will have a different fade value, so they'll all burn out at different speeds.

particle[loop].life-=particle[loop].fade; // Reduce

Now we check to see if the particle is still alive after having life taken from it.

```
if (particle[loop].life<0.0f)
{</pre>
```

If the particle is dead (burnt out), we'll rejuvenate it. We do this by giving it full life and a new fade speed.

particle[loop].life=1.0f; particle[loop].fade=float(rand()%100)/1000.0f+0.003f;

We also reset the particles position to the center of the screen. We do this by resetting the x, y and z positions of the particle to zero.

particle[loop].x=0.0f; particle[loop].y=0.0f; particle[loop].z=0.0f; After the particle has been reset to the center of the screen, we give it a new moving speed / direction. Notice I've increased the maximum and minimum speed that the particle can move at from a random value of 50 to a value of 60, but this time we're not going to multiply the moving speed by 10. We don't want an explosion this time around, we want slower moving particles.

Also notice that I add **xspeed** to the x axis moving speed, and **yspeed** to the y axis moving speed. This gives us control over what direction the particles move later in the program.

```
particle[loop].xi=xspeed+float((rand()%60)-32.0f);
particle[loop].yi=yspeed+float((rand()%60)-30.0f);
particle[loop].zi=float((rand()%60)-30.0f);
```

Lastly we assign the particle a new color. The variable **col** holds a number from 0 to 11 (12 colors). We use this variable to look of the red, green and blue intensities in our color table that we made at the beginning of the program. The first line below sets the red (**r**) intensity to the red value stored in **colors[col][0]**. So if col was 0, the red intensity would be 1.0f. The green and blue values are read the same way.

If you don't understand how I got the value of 1.0f for the red intensity if col is 0, I'll explain in a bit more detail. Look at the very top of the program. Find the line: static GLfloat colors[12][3]. Notice there are 12 groups of 3 number. The first of the three number is the red intensity. The second value is the green intensity and the third value is the blue intensity. [0], [1] and [2] below represent the 1st, 2nd and 3rd values I just mentioned. If **col** is equal to 0, we want to look at the first group. 11 is the last group (12th color).

```
particle[loop].r=colors[col][0];
particle[loop].g=colors[col][1];
particle[loop].b=colors[col][2];
```

The line below controls how much gravity there is pulling upward. By pressing 8 on the number pad, we increase the **yg** (y gravity) variable. This causes a pull upwards. This code is located here in the program because it makes our life easier by applying the gravity to all of our particles thanks to the loop. If this code was outside the loop we'd have to create another loop to do the same job, so we might as well do it here.

}

// If Number Pad 8 And Y Gravity Is Less Than 1.5 Increase Pull if (keys[VK_NUMPAD8] && (particle[loop].yg<1.5f)) particle[loop]</pre>

This line has the exact opposite affect. By pressing 2 on the number pad we decrease **yg** creating a stronger pull downwards.

// If Number Pad 2 And Y Gravity Is Greater Than -1.5 Increase P
if (keys[VK_NUMPAD2] && (particle[loop].yg>-1.5f)) particle[loop

Now we modify the pull to the right. If the 6 key on the number pad is pressed, we increase the pull to the right.

// If Number Pad 6 And X Gravity Is Less Than 1.5 Increase Pull if (keys[VK_NUMPAD6] && (particle[loop].xg<1.5f)) particle[loop]</pre>

Finally, if the 4 key on the number pad is pressed, our particle will pull more to the left. These keys give us some really cool results. For example, you can make a stream of particles shooting straight up in the air. By adding some gravity pulling downwards you can turn the stream of particles into a fountain of water!

// If Number Pad 4 And X Gravity Is Greater Than -1.5 Increase P
if (keys[VK_NUMPAD4] && (particle[loop].xg>-1.5f)) particle[loop

I added this bit of code just for fun. My brother thought the explosion was a cool effect :) By pressing the tab key all the particles will be reset back to the center of the screen. The moving speed of the particles will once again be multiplied by 10, creating a big explosion of particles. After the particles fade out, your original effect will again reappear.

The code in KillGLWindow(), CreateGLWindow() and WndProc() hasn't changed, so we'll skip down to WinMain(). I'll rewrite the entire section of code to make it easier to follow through the code.

```
// Instance
int WINAPI WinMain(
                          HINSTANCEhInstance,
                          HINSTANCEhPrevInstance,
                                                                       // Previous Instand
                                                                                // Comman
                          LPSTR
                                         lpCmdLine,
                                                                       // Window Show Stat
                                           nCmdShow)
                          int
{
                                                                                // Window
        MSG
                 msg;
                                                                                // Bool Va
        BOOL
                 done=FALSE;
         // Ask The User Which Screen Mode They Prefer
        if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreen"
         {
                 fullscreen=FALSE;
                                                                       // Windowed Mode
         }
```

This is our first change to WinMain(). I've added some code to check if the user decide to run in fullscreen mode or windowed mode. If they decide to use fullscreen mode, I change the variable **slowdown** to 1.0f instead of 2.0f. You can leave this bit code out if you want. I added the code to speed up fullscreen mode on my 3dfx (runs ALOT slower than windowed mode for some reason).

```
if (fullscreen)
                                                                            // Are We
{
         slowdown=1.0f;
                                                                            // Speed 1
}
while(!done)
                                                                            // Loop T
{
         if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
                                                                  // Is There A Messa
         {
                   if (msq.message==WM_QUIT)
                                                                  // Have We Received
                   {
                            done=TRUE;
                                                                            // If So (
                   }
                                                                            // If Not
                   else
                   {
                                                                            // Transla
                            TranslateMessage(&msg);
                            DispatchMessage(&msg);
                                                                            // Dispat
                   }
         }
                                                                            // If The:
         else
         {
                   if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
                                                                            // Updati
                   {
                            done=TRUE;
                                                                            // ESC or
                   }
                   else
                                                                            // Not Tii
                   {
                            SwapBuffers(hDC);
                                                                // Swap Buffers (Do
```

I was a little sloppy with the next bit of code. Usually I don't include everything on one line, but it makes the code look a little cleaner :)

The line below checks to see if the + key on the number pad is being pressed. If it is and **slowdown** is greater than 1.0f we decrease **slowdown** by 0.01f. This causes the particles to move faster. Remember in the code above when I talked about **slowdown** and how it affects the speed at which the particles travel.

if (keys[VK_ADD] && (slowdown>1.0f)) slowdown=0.01f;

This line checks to see if the - key on the number pad is being pressed. If it is and **slowdown** is less than 4.0f we increase the value of **slowdown**. This causes our particles to move slower. I put a limit of 4.0f because I wouldn't want them to move much slower. You can change the minimum and maximum speeds to whatever you want :)

if (keys[VK_SUBTRACT] && (slowdown<4.0f)) slowdown+=0.0

The line below check to see if Page Up is being pressed. If it is, the variable **zoom** is increased. This causes the particles to move closer to us.

if (keys[VK_PRIOR]) zoom+=0.1f; // Zoom I:

This line has the opposite effect. By pressing Page Down, **zoom** is decreased and the scene moves futher into the screen. This allows us to see more of the screen, but it makes the particles smaller.

if (keys[VK_NEXT]) zoom-=0.1f; // Zoom O

The next section of code checks to see if the return key has been pressed. If it has and it's not being 'held' down, we'll let the computer know it's being pressed by setting rp to true. Then we'll toggle rainbow mode. If **rainbow** was true, it will become false. If it was false, it will become true. The last line checks to see if the return key was released. If it was, **rp** is set to false, telling the computer that the key is no longer being held down.

if (keys[VK_RETURN] && !rp)	// Return Key Pres:
{ rp=true;	// Set Flag Telling
rainbow=!rainbow;	// Toggle Rainbow 1
<pre> f (!keys[VK_RETURN]) rp=false; </pre>	// If Ret

The code below is a little confusing. The first line checks to see if the spacebar is being pressed and not held down. It also check to see if rainbow mode is on, and if so, it checks to see if the variable **delay** is greater than 25. **delay** is a counter I use to create the rainbow effect. If you were to change the color ever frame, the particles would all be a different color. By creating a delay, a group of particles will become one color, before the color is changed to something else.

If the spacebar was pressed or rainbow is on and **delay** is greater than 25, the color will be changed!

if ((keys[' '] && !sp) || (rainbow && (delay>25))) {

The line below was added so that rainbow mode would be turned off if the spacebar was pressed. If we didn't turn off rainbow mode, the colors would continue cycling until the return key was pressed again. It makes sense that if the person is hitting space instead of return that they want to go through the colors themselves.

```
if (keys[' ']) rainbow=false; // If Spa
```

If the spacebar was pressed or rainbow mode is on, and **delay** is greater than 25, we'll let the computer know that space has been pressed by making **sp** equal true. Then we'll set the delay back to 0 so that it can start counting back up to 25. Finally we'll increase the variable **col** so that the color will change to the next color in the color table.

sp=true;// Set Flag Tellingdelay=0;// Reset The Rainbocol++;// Change

If the color is greater than 11, we reset it back to zero. If we didn't reset **col** to zero, our program would try to find a 13th color. We only have 12 colors! Trying to get information about a color that doesn't exist would crash our program.

```
if (col>11) col=0; // If Color Is To 1
}
```

Lastly if the spacebar is no longer being pressed, we let the computer know by setting the variable **sp** to false.

if (!keys[' ']) sp=false; // If Spacebar Is I

Now for some control over the particles. Remember that we created 2 variables at the beginning of our program? One was called **xspeed** and one was called **yspeed**. Also remember that after the particle burned out, we gave it a new moving speed and added the new speed to either **xspeed** or **yspeed**. By doing that we can influence what direction the particles will move when they're first created.

For example. Say our particle had a moving speed of 5 on the x axis and 0 on the y axis. If we decreased **xspeed** until it was -10, we would be moving at a speed of -10 (**xspeed**) + 5 (original moving speed). So instead of moving at a rate of 10 to the right we'd be moving at a rate of -5 to the left. Make sense?

Anyways. The line below checks to see if the up arrow is being pressed. If it is, **yspeed** will be increased. This will cause our particles to move upwards. The particles will move at a maximum speed of 200 upwards. Anything faster than that doesn't look to good.

// If Up Arrow And Y Speed Is Less Than 200 Increase Up
if (keys[VK_UP] && (yspeed<200)) yspeed+=1.0f;</pre>

}

This line checks to see if the down arrow is being pressed. If it is, **yspeed** will be decreased. This will cause the particles to move downward. Again, a maximum downward speed of 200 is enforced.

// If Down Arrow And Y Speed Is Greater Than -200 Incre
if (keys[VK_DOWN] && (yspeed>-200)) yspeed-=1.0f;

Now we check to see if the right arrow is being pressed. If it is, **xspeed** will be increased. This will cause the particles to move to the right. A maximum speed of 200 is enforced.

// If Right Arrow And X Speed Is Less Than 200 Increase
if (keys[VK_RIGHT] && (xspeed<200)) xspeed+=1.0f;</pre>

Finally we check to see if the left arrow is being pressed. If it is... you guessed it... **xspeed** is decreased, and the particles start to move left. Maximum speed of 200 enforced.

```
// If Left Arrow And X Speed Is Greater Than -200 Incre
if (keys[VK_LEFT] && (xspeed>-200)) xspeed-=1.0f;
```

The last thing we need to do is increase the variable **delay**. Like I said above, **delay** is used to control how fast the colors change when you're using rainbow mode.

delay++; // Increase Rainbow Mode Col

Like all the previous tutorials, make sure the title at the top of the window is correct.

```
// Is F1 Being Pressed?
                           if (keys[VK_F1])
                           {
                                   keys[VK_F1]=FALSE; // If So Make Key FALSE
                                   KillGLWindow();
                                                              // Kill Our Current
                                                              // Toggle Fullscre
                                   fullscreen=!fullscreen;
                                    // Recreate Our OpenGL Window
                                   if (!CreateGLWindow("NeHe's Particle Tutorial"
                                    {
                                            return 0;// Quit If Window Was Not Ci
                                    }
                           }
                 }
         }
}
// Shutdown
                                                              // Kill The Window
KillGLWindow();
                                                              // Exit The Program
return (msg.wParam);
```

In this lesson, I have tried to explain in as much detail, all the steps required to create a simple but impressive particle system. This particle system can be used in games of your own to create effects such as Fire, Water, Snow, Explosions, Falling Stars, and more. The code can easily be modified to handle more parameters, and new effects (fireworks for example).

Thanks to Richard Nutman for suggesting that the particles be positioned with glVertex3f() instead of resetting the Modelview Matrix and repositioning each particle with glTranslatef(). Both methods are effective, but his method will reduce the amount of work the computer has to do before it draws each particle, causing the program to run even faster.

Thanks to Antoine Valentim for suggesting triangle strips to help speed up the program and to introduce a new command to this tutorial. The feedback on this tutorial has been great, I appreciate it!

I hope you enjoyed this tutorial. If you had any problems understanding it, or you've found a mistake in the tutorial please let me know. I want to make the best tutorials available. Your feedback is important!

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Owen Borstad)
- * DOWNLOAD Irix Code For This Lesson. (Conversion by Dimitrios Christopoulos)

Back To NeHe Productions!

Lesson 20

Welcome to Tutorial 20. The bitmap image format is supported on just about every computer, and just about every operating system. Not only is it easy to work with, it's very easy to load and use as a texture. Up until now, we've been using blending to place text and other images onto the screen without erasing what's underneath the text or image. This is effective, but the results are not always pretty.

Most the time a blended texture blends in too much or not enough. When making a game using sprites, you don't want the scene behind your character shining through the characters body. When writing text to the screen you want the text to be solid and easy to read.

That's where masking comes in handy. Masking is a two step process. First we place a black and white image of our texture on top of the scene. The white represents the transparent part of our texture. The black represents the solid part of our texture. Because of the type of blending we use, only the black will appear on the scene. Almost like a cookie cutter effect. Then we switch blending modes, and map our texture on top of the black cut out. Again, because of the blending mode we use, the only parts of our texture that will be copied to the screen are the parts that land on top of the black mask.

I'll rewrite the entire program in this tutorial aside from the sections that haven't changed. So if you're ready to learn something new, let's begin!

<pre>#include #include #include #include</pre>	<windows.h> <math.h> <stdio.h> <gl\gl.h> <gl\glu.h> <gl\glaux.h></gl\glaux.h></gl\glu.h></gl\gl.h></stdio.h></math.h></windows.h>	<pre>// Header File For Windows // Header File For Windows Math Libra // Header File For Standard Input/Out // Header File For The OpenGL32 Libra // Header File For The GLu3; // Header File For The Glaux</pre>
HDC	hDC=NULL;	// Private GDI Device Context
HGLRC	hRC=NULL;	// Permanent Rendering Context
HWND	hWnd=NULL;	// Holds Our Window Handle
HINSTANCH	chInstance;	// Holds The Instance Of The Applicat

We'll be using 7 global variables in this program. **masking** is a boolean variable (TRUE / FALSE) that will keep track of whether or not masking is turned on of off. **mp** is used to make sure that the 'M' key isn't being held down. **sp** is used to make sure that the 'Spacebar' isn't being held down and the variable **scene** will keep track of whether or not we're drawing the first or second scene.

We set up storage space for 5 textures using the variable **texture**[5]. **loop** is our generic counter variable, we'll use it a few times in our program to set up textures, etc. Finally we have the variable **roll**. We'll use **roll** to roll the textures across the screen. Creates a neat effect! We'll also use it to spin the object in scene 2.

bool keys[256]; bool active=TRUE; bool fullscreen=TRUE; bool masking=TRUE;

// Array Used For The Keyboa // Window Active Flag Set Tc // Fullscreen Flag Set To Fullscreen // Masking On/Off

Jeff Molofee's OpenGL Windows Tutorial #20

bool	mp;	// M Pressed?
bool	sp;	// Space Pressed?
bool	scene;	// Which Scene To Draw
GLuint	<pre>texture[5];</pre>	// Storage For Our Five Text
GLuint	loop;	// Generic Loop Variable
GLfloat	roll;	// Rolling Texture
LRESULT	CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Decla	ration For WndProc

The load bitmap code hasn't changed. It's the same as it was in lesson 6, etc.

In the code below we create storage space for 5 images. We clear the space and load in all 5 bitmaps. We loop through each image and convert it into a texture for use in our program. The textures are stored in **texture[0-4]**.

```
// Load B
int LoadGLTextures()
         int Status=FALSE;
                                                                           // Status Indicato
         AUX_RGBImageRec *TextureImage[5];
                                                                           // Create Storage :
         memset(TextureImage,0,sizeof(void *)*5);
                                                                           // Set The Pointer
         if ((TextureImage[0]=LoadBMP("Data/logo.bmp")) &&
                                                                           // Logo Texture
             (TextureImage[1]=LoadBMP("Data/mask1.bmp")) &&
                                                                           // First Mask
             (TextureImage[2]=LoadBMP("Data/image1.bmp")) &&
                                                                          // First Image
             (TextureImage[3]=LoadBMP("Data/mask2.bmp")) &&
                                                                          // Second Mask
             (TextureImage[4]=LoadBMP("Data/image2.bmp")))
                                                                           // Second Image
         {
                  Status=TRUE;
                                                                                    // Set The
                  glGenTextures(5, &texture[0]);
                                                                                    // Create
                  for (loop=0; loop<5; loop++)</pre>
                                                                                    // Loop T
                   {
                            glBindTexture(GL_TEXTURE_2D, texture[loop]);
                            glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                            glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
                            glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, Tex
                                     0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
                  }
         }
         for (loop=0; loop<5; loop++)</pre>
                                                                                    // Loop T
         {
                  if (TextureImage[loop])
                                                                                    // If Tex
                   {
                            if (TextureImage[loop]->data)
                                                                                    // If Tex
                            {
                                     free(TextureImage[loop]->data);
                                                                                    // Free Tl
                            }
                            free(TextureImage[loop]);
                                                                           // Free The Image :
                  }
         }
                                                                                    // Return
         return Status;
}
```

The ReSizeGLScene() code hasn't changed so we'll skip over it.

The Init code is fairly bare bones. We load in our textures, set the clear color, set and enable depth testing, turn on smooth shading, and enable texture mapping. Simple program so no need for a complex init :)

```
int InitGL(GLvoid)
                                                                                     // All Se
{
         if (!LoadGLTextures())
         {
                  return FALSE;
         }
         glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
         glClearDepth(1.0);
                                                                                     // Enable
         glEnable(GL_DEPTH_TEST);
                                                                                     // Enable
                                                                                     // Enable
         glShadeModel(GL_SMOOTH);
                                                                                     // Enable
         glEnable(GL_TEXTURE_2D);
         return TRUE;
}
```

Now for the fun stuff. Our drawing code! We start off the same as usual. We clear the background color and the depth buffer. Then we reset the modelview matrix, and translate into the screen 2 units so that we can see our scene.

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear '
    glLoadIdentity(); // Reset '
    glTranslatef(0.0f,0.0f,-2.0f);
```

The first line below select the 'logo' texture. We'll map the texture to the screen using a quad. we specify our 4 texture coordinates along with our 4 vertices.

You'll notice that the texture coordinates may look weird. Instead of using 1.0 and 0.0 I'm using 3.0 and 0.0. I'll explain what this does. By using 3.0 as a texture coordinate instead of 1.0, we are telling OpenGL to draw our texture 3 times. Normally our one texture is mapped across the entire face of our quad. This time OpenGL will squish 3 of our textures onto the quad. I'm also using 3.0 for the up and down value, meaning we'll have three textures wide, and 3 textures up and down. Mapping 9 images of the selected texture to the front of our quad.

		N	\sim
•	•	•	I
-	-	н	
0	0	0	V
N	N	N	_
•	•	•	S
-	-	- 14	
•	•	•	p
2	2	Z	i.
•	•	•	h
-	-	- 14	4
•	•	•	1

You will also notice that I've added the variable **-roll** to our verticle texture coordinates. This cause the texture to roll up the screen as the value of **roll** increases. **roll** tells OpenGL what part of our image to start texturing from. The image to the left is how our texture would look if **roll** was equal to 0.0. From 0.0 to 1.0 would be the first texture drawn up and down. From 1.0 to



2.0 would be our second texture, and from 2.0 to 3.0 would be our third texture. The image on the right shows how our texture would look if **roll** was equal to 0.5. Our first texture would be drawn from -0.5 to 0.5 (notice that because we started drawing halfway through the texture that the 'N' and 'e' have been cut off). The second texture would be from 0.5 to 1.5, and the third texture would be from 1.5 to 2.5. Again notice that only the 'N' and 'e' have been drawn at the bottom. We never quite made it to 3.0 (the bottom of a complete texture) so the 'H' and 'e' were not drawn. Rolling textures can be used to create great effects such as moving clouds. Words spinning around an object, etc.

If you don't understand what I mean about rolling textures, let me know. If you have a better way to explain let me know. It's easy to understand how rolling textures work once you've used them, but trying to explain it in words isn't very easy.

One last explanation to hopefully clear things up. Imagine you had an endless amount of marbles up and down, left and right. Every marble was identicle (imagine each marble is a texture). The marble in the center of your infinate number of marbles is your main marble (texture). It's left side is 0.0, it's right side is 1.0, the values up and down are also 0.0 to 1.0. Now if you move left half a marble (-0.5), and you can only see 1.0 marbles wide you would only see the right half of the marble to the left of your original marble and the left half of your original marble. If you moved left another half (-0.5... a total of -1.0) you would see an entire marble (texture) but it wouldn't be your original marble, it would be the marble to the left of it. Because all the marbles look exactly the same you would think you were seeing your entire original marble (texture). {grin}. Hopefully that doesn't confuse you even more. I know how some of you hate my little stories.

Anyways... back to reality. Now we enable blending. In order for this effect to work we also have to disable depth testing. It's very important that you do this! If you do not disable depth testing you probably wont see anything. Your entire image will vanish!

glEnable(GL_BLEND);
glDisable(GL_DEPTH_TEST);

// Disabl

The first thing we do after we enable blending and disable depth testing is check to see if we're going to mask our image or blend it the old fashioned way. The line of code below checks to see if **masking** is TRUE. If it is we'll set up blending so that our mask gets drawn to the screen properly.

```
if (masking)
{
```

If **masking** is TRUE the line below will set up blending for our mask. A mask is just a copy of the texture we want to draw to the screen but in black and white. Any section of the mask that is white will be transparent. Any sections of the mask that is black will be SOLID.

The blend command below does the following: The Destination color (screen color) will be set to black if the section of our mask that is being copied to the screen is black. This means that sections of the screen that the black portion of our mask covers will turn black. Anything that was on the screen under the mask will be cleared to black. The section of the screen covered by the white mask will not change.

// Blend :

Now we check to see what scene to draw. If **scene** is TRUE we will draw the second scene. If **scene** is FALSE we will draw the first scene.

```
if (scene)
{
```

We don't want things to be too big so we translate one more unit into the screen. This reduces the size of our objects.

After we translate into the screen, we rotate from 0-360 degrees depending on the value of **roll**. If **roll** is 0.0 we will be rotating 0 degrees. If **roll** is 1.0 we will be rotating 360 degrees. Fairly fast rotation, but I didn't feel like creating another variable just to rotate the image in the center of the screen. :)

```
glTranslatef(0.0f,0.0f,-1.0f);
glRotatef(roll*360,0.0f,0.0f,1.0f);
```

// Rotate

We already have the rolling logo on the screen and we've rotated the scene on the Z axis causing any objects we draw to be rotated counter-clockwise, now all we have to do is check to see if masking is on. If it is we'll draw our mask then our object. If masking is off we'll just draw our object.

```
if (masking)
{
```

If **masking** is TRUE the code below will draw our mask to the screen. Our blend mode should be set up properly because we had checked for masking once already while setting up the blending. Now all we have to do is draw the mask to the screen. We select mask 2 (because this is the second scene). After we have selected the mask texture we texture map it onto a quad. The quad is 1.1 units to the left and right so that it fills the screen up a little more. We only want one texture to show up so our texture coordinates only go from 0.0 to 1.0.

after drawing our mask to the screen a solid black copy of our final texture will appear on the screen. The final result will look as if someone took a cookie cutter and cut the shape of our final texture out of the screen, leaving an empty black space.

Now that we have drawn our mask to the screen it's time to change blending modes again. This time we're going to tell OpenGL to copy any part of our colored texture that is NOT black to the screen. Because the final texture is an exact copy of the mask but with color, the only parts of our texture that get drawn to the screen are parts that land on top of the black portion of the mask. Because the mask is black, nothing from the screen will shine through our texture. This leaves us with a very solid looking texture floating on top of the screen.

Notice that we select the second image after selecting the final blending mode. This selects our colored image (the image that our second mask is based on). Also notice that we draw this image right on top of the mask. Same texture coordinates, same vertices.

If we don't lay down a mask, our image will still be copied to the screen, but it will blend with whatever was on the screen.

If scene was FALSE, we will draw the first scene (my favorite).

```
else
{
```

We start off by checking to see if **masking** is TRUE of FALSE, just like in the code above.

```
if (masking)
{
```

}

If **masking** is TRUE we draw our mask 1 to the screen (the mask for scene 1). Notice that the texture is rolling from right to left (**roll** is added to the horizontal texture coordinate). We want this texture to fill the entire screen that is why we never translated further into the screen.

Again we enable blending and select our texture for scene 1. We map this texture on top of it's mask. Notice we roll this texture as well, otherwise the mask and final image wouldn't line up.

Next we enable depth testing, and disable blending. This prevents strange things from happening in the rest of our program :)

```
glEnable(GL_DEPTH_TEST);
glDisable(GL_BLEND);
```

// Enable

Finally all we have left to do is increase the value of **roll**. If **roll** is greater than 1.0 we subtract 1.0. This prevents the value of **roll** from getting to high.

```
roll+=0.002f;
if (roll>1.0f)
{
            roll-=1.0f;
}
return TRUE;
```

}

The KillGLWindow(), CreateGLWindow() and WndProc() code hasn't changed so we'll skip over it.

The first thing you will notice different in the WinMain() code is the Window title. It's now titled "NeHe's Masking Tutorial". Change it to whatever you want :)

```
int WINAPI WinMain(
                          HINSTANCEhInstance,
                                                                                // Instan
                          HINSTANCEhPrevInstance,
                                                                                // Previo
                         LPSTR lpCmdLine,
                          int
                                          nCmdShow)
                                                                                // Window
{
        MSG
                 msg;
        BOOL
                 done=FALSE;
        // Ask The User Which Screen Mode They Prefer
        if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreen"
        {
                 fullscreen=FALSE;
                                                                                // Window
        }
```

```
Jeff Molofee's OpenGL Windows Tutorial #20
```

```
// Create Our OpenGL Window
if (!CreateGLWindow("NeHe's Masking Tutorial",640,480,16,fullscreen))
{
         return 0;
                                                                           // Quit I:
}
while(!done)
{
         if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
                                                                           // Is The:
         {
                  if (msg.message==WM_QUIT)
                                                                           // Have W
                   {
                            done=TRUE;
                  }
                  else
                   {
                            TranslateMessage(&msg);
                            DispatchMessage(&msg);
                   }
         }
         else
         {
                   // Draw The Scene. Watch For ESC Key And Quit Messages From Dra
                  if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active
                   {
                            done=TRUE;
                  }
                  else
                   {
                            SwapBuffers(hDC);
                                                                           // Swap B
```

Now for our simple key handling code. We check to see if the spacebar is being pressed. If it is, we set the **sp** variable to TRUE. If **sp** is TRUE, the code below will not run a second time until the spacebar has been released. This keeps our program from flipping back and forth from scene to scene very rapidly. After we set **sp** to TRUE, we toggle the scene. If it was TRUE, it becomes FALSE, if it was FALSE it becomes TRUE. In our drawing code above, if **scene** is FALSE the first scene is drawn. If **scene** is TRUE the second scene is drawn.

The code below checks to see if we have released the spacebar (if NOT ' '). If the spacebar has been released, we set **sp** to FALSE letting our program know that the spacebar is NOT being held down. By setting **sp** to FALSE the code above will check to see if the spacebar has been pressed again, and if so the cycle will start over.

```
if (!keys[' '])
{
            sp=FALSE; // Tell P:
}
```

}

The next section of code checks to see if the 'M' key is being pressed. If it is being pressed, we set **mp** to TRUE, telling our program not to check again until the key is released, and we toggle **masking** from TRUE to FALSE or FALSE to TRUE. If **masking** is TRUE, the drawing code will turn on masking. If it is FALSE masking will be off. If masking is off, the object will be blended to the screen using the old fashioned blending we've been using up until now.

The last bit of code checks to see if we've stopped pressing 'M'. If we have, **mp** becomes FALSE letting the program know that we are no longer holding the 'M' key down. Once the 'M' key has been released, we are able to press it once again to toggle masking on or off.

```
if (!keys['M'])
{
     mp=FALSE; // Tell P:
}
```

Like all the previous tutorials, make sure the title at the top of the window is correct.

```
if (keys[VK_F1])
                                                   // Is F1 Being Pressed?
                          {
                                   keys[VK_F1]=FALSE; // If So Make Key FALSE
                                   KillGLWindow(); // Kill Our Current
                                   fullscreen=!fullscreen;
                                                            // Toggle Fullscre
                                   // Recreate Our OpenGL Window
                                   if (!CreateGLWindow("NeHe's Masking Tutorial",
                                   {
                                           return 0;// Quit If Window Was Not Cu
                                   }
                          }
                 }
         }
}
// Shutdown
                                                             // Kill The Window
KillGLWindow();
return (msg.wParam);
                                                             // Exit The Program
```

Creating a mask isn't to hard. A little time consuming. The best way to make a mask if you already have your image made is to load your image into an art program or a handy program like infranview, and reduce it to a gray scale image. After you've done that, turn the contrast way up so that gray pixels become black. You can also try turning down the brightness, etc. It's important that the white is bright white, and the black is pure black. If you have any gray pixels in your mask, that section of the image will appear transparent. The most reliable way to make sure your mask is a perfect copy of your image is to trace over the image with black. It's also very important that your image has a BLACK background and the mask has a WHITE background! If you create a mask and notice a square shape around your texture, either your white isn't bright enough (255 or FFFFFF) or your black isn't true black (0 or 000000). Below you can see an example of a mask and the image that goes over top of the mask. the image can be any color you want as long as the background is black. The mask must have a white background and a black copy of your image.

This is the mask ->



This is the image ->



Eric Desrosiers pointed out that you can also check the value of each pixel in your bitmap while you load it. If you want the pixel transparent you can give it an alpha value of 0. For all the other colors you can give them an alpha value of 255. This method will also work but requires some extra coding. The current tutorial is simple and requires very little extra code. I'm not blind to other techniques, but when I write a tutorial I try to make the code easy to understand and easy to use. I just wanted to point out that there are always other ways to get the job done. Thanks for the feedback Eric.

In this tutorial I have shown you a simple, but effective way to draw sections of a texture to the screen without using the alpha channel. Normal blending usually looks bad (textures are either transparent or they're not), and texturing with an alpha channel requires that your images support the alpha channel. Bitmaps are convenient to work with, but they do not support the alpha channel this program shows us how to get around the limitations of bitmap images, while demonstrating a cool way to create overlay type effects.

Thanks to Rob Santa for the idea and for example code. I had never heard of this little trick until he pointed it out. He wanted me to point out that although this trick does work, it takes two passes, which causes a performance hit. He recommends that you use textures that support the alpha channel for complex scenes.

I hope you enjoyed this tutorial. If you had any problems understanding it, or you've found a mistake in the tutorial please let me know. I want to make the best tutorials available. Your feedback is important!

Jeff Molofee (NeHe)

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Anthony Parker)

Back To NeHe Productions!

Lesson 21

Welcome to my 21st OpenGL Tutorial! Coming up with a topic for this tutorial was extremely difficult. I know alot of you are tired of learning the basics. Everyone is dying to learn about 3D objects, Multitexturing and all that other good stuff. For those people, I'm sorry, but I want to keep the learning curve gradual. Once I've gone a step ahead it's not as easy to take a step back without people losing interest. So I'd prefer to keep pushing forward at a steady pace.

In case I've lost a few of you :) I'll tell you a bit about this tutorial. Until now all of my tutorials have used polygons, quads and triangles. So I decided it would be nice to write a tutorial on lines. A few hours after starting the line tutorial, I decided to call it quits. The tutorial was coming along fine, but it was BORING! Lines are great, but there's only so much you can do to make lines exciting. I read through my email, browsed through the message board, and wrote down a few of your tutorial requests. Out of all the requests there were a few questions that came up more than others. So... I decided to write a multi-tutorial :)

In this tutorial you will learn about: Lines, Anti-Aliasing, Orthographic Projection, Timing, Basic Sound Effects, and Simple Game Logic. Hopefully there's enough in this tutorial to keep everyone happy :) I spent 2 days coding this tutorial, and It's taken almost 2 weeks to write this HTML file. I hope you enjoy my efforts!

At the end of this tutorial you will have made a simple 'amidar' type game. Your mission is to fill in the grid without being caught by the bad guys. The game has levels, stages, lives, sound, and a secret item to help you progress through the levels when things get tough. Although this game will run fine on a Pentium 166 with a Voodoo 2, a faster processor is recommended if you want smoother animation.

I used the code from lesson 1 as a starting point while writing this tutorial. We start off by adding the required header files. stdio.h is used for file operations, and we include stdarg.h so that we can display variables on the screen, such as the score and current stage.

/* * */	This Code Was Created By Jeff Molofee 2000 If You've Found This Code Useful, Please Let Me Know.	
<pre>#include <windows #include="" <gl\gl.h="" <gl\glau<="" <gl\glu.="" <stdarg.="" <stdio.h="" pre=""></windows></pre>	> h> > h>	// Header // Standard Input , // Header // Header File For // Header // Header
HDC HGLRC HWND HINSTANCEhInstanc	hDC=NULL; hRC=NULL; hWnd=NULL; e;	// Privat // Perman // Holds '

Now we set up our boolean variables. **vline** keeps track of the 121 vertical lines that make up our game grid. 11 lines across and 11 up and down. **hline** keeps track of the 121 horizontal lines that make up the game grid. We use **ap** to keep track of whether or not the 'A' key is being pressed.

filled is FALSE while the grid isn't filled and TRUE when it's been filled in. **gameover** is pretty obvious. If **gameover** is TRUE, that's it, the game is over, otherwise you're still playing. **anti** keeps track of antialiasing. If **anti** is TRUE, object antialiasing is ON. Otherwise it's off. **active** and **fullscreen** keep track of whether or not the program has been minimized or not, and whether you're running in fullscreen mode or windowed mode.

bool	keys[256];	
bool	vline[11][10];	
bool	hline[10][11];	
bool	ap;	
bool	filled;	
bool	gameover;	// Is The
bool	anti=TRUE;	
bool	active=TRUE;	
bool	fullscreen=TRUE;	// Fullsc:

Now we set up our integer variables. **loop1** and **loop2** will be used to check points on our grid, see if an enemy has hit us and to give objects random locations on the grid. You'll see **loop1** / **loop2** in action later in the program. **delay** is a counter variable that I use to slow down the bad guys. If **delay** is greater than a certain value, the enemies are moved and **delay** is set back to zero.

The variable **adjust** is a very special variable! Even though this program has a timer, the timer only checks to see if your computer is too fast. If it is, a delay is created to slow the computer down. On my GeForce card, the program runs insanely smooth, and very very fast. After testing this program on my PIII/450 with a Voodoo 3500TV, I noticed that the program was running extremely slow. The problem is that my timing code only slows down the gameplay. It wont speed it up. So I made a new variable called **adjust**. **adjust** can be any value from 0 to 5. The objects in the game move at different speeds depending on the value of adjust. The lower the value the smoother they move, the higher the value, the faster they move (choppy at values higher than 3). This was the only real easy way to make the game playable on slow systems. One thing to note, no matter how fast the objects are moving the game speed will never run faster than I intended it to run. So setting the **adjust** value to 3 is safe for fast and slow systems.

The variable **lives** is set to 5 so that you start the game with 5 lives. **level** is an internal variable. The game uses it to keep track of the level of difficulty. This is not the level that you will see on the screen. The variable **level2** starts off with the same value as **level** but can increase forever depending on your skill. If you manage to get past level 3 the **level** variable will stop increasing at 3. The **level** variable is an internal variable used for game difficulty. The **stage** variable keeps track of the current game stage.

int int	loop1; loop2; delay;
int	adjust=3;
int	lives=5;
int	<pre>level=1;</pre>
int	level2=level;
int	stage=1;

// Speed 1
// Player
// Interna

// Game S

Now we create a structure to keep track of the objects in our game. We have a fine X position (**fx**) and a fine Y position (**fy**). These variables will move the player and enemies around the grid a few pixels at a time. Creating a smooth moving object.

Then we have \mathbf{x} and \mathbf{y} . These variables will keep track of what intersection our player is at. There are 11 points left and right and 11 points up and down. So \mathbf{x} and \mathbf{y} can be any value from 0 to 10. That is why we need the fine values. If we could only move one of 11 spots left and right and one of 11 spots up and down our player would jump around the screen in a quick (non smooth) motion.

The last variable **spin** will be used to spin the objects on their z-axis.

struct {	object	
};	int int float	fx, fy; x, y; spin;

Now that we have created a structure that can be used for our player, enemies and even a special item we can create new structures that take on the characteristics of the structure we just made.

The first line below creates a structure for our player. Basically we're giving our player structure **fx**, **fy**, **x**, **y** and **spin** values. By adding this line, we can access the player **x** position by checking **player.x**. We can change the player spin by adding a number to **player.spin**.

The second line is a bit different. Because we can have up to 15 enemies on the screen at a time, we need to create the above variables for each enemy. We do this by making an array of 15 enemies. the **x** position of the first enemy will be **enemy[0].x**. The second enemy will be **enemy [1].x**, etc.

The last line creates a structure for our special item. The special item is an hourglass that will appear on the screen from time to time. We need to keep track of the x and y values for the hourglass, but because the hourglass doesn't move, we don't need to keep track of the fine positions. Instead we will use the fine variables (fx and fy) for other things later in the program.

struct	object	player;
struct	object	enemy[9];
struct	object	hourglass;

// Enemy

Now we create a timer structure. We create a structure so that it's easier to keep track of timer variables and so that it's easier to tell that the variable is a timer variable.

The first thing we do is create a 64 bit integer called **frequency**. This variable will hold the frequency of the timer. When I first wrote this program, I forgot to include this variable. I didn't realize that the frequency on one machine may not match the frequency on another. Big mistake on my part! The code ran fine on the 3 systems in my house, but when I tested it on a friends machine the game ran WAY to fast. Frequency is basically how fast the clock is updated. Good thing to keep track of :)

The **resolution** variable keeps track of the steps it takes before we get 1 millisecond of time.

mm_timer_start and mm_timer_elapsed hold the value that the timer started at, and the amount

of time that has elapsed since the timer was started. These two variables are only used if the computer doesn't have a performance counter. In that case we end up using the less accurate multimedia timer, which is still not to bad for a non-time critical game like this.

The variable **performance_timer** can be either TRUE of FALSE. If the program detects a performance counter, the variable **performance_timer** variable is set to TRUE, and all timing is done using the performance counter (alot more accurate than the multimedia timer). If a performance counter is not found, **performance_timer** is set to FALSE and the multimedia timer is used for timing.

The last 2 variables are 64 bit integer variables that hold the start time of the performance counter and the amount of time that has elapsed since the performance counter was started.

The name of this structure is "timer" as you can see at the bottom of the structure. If we want to know the timer frequency we can now check **timer.frequency**. Nice!

struct

int64	frequency;	//	Timer :
float	resolution;	//	Timer 1
unsigned long	mm_timer_start;		
unsigned long	mm_timer_elapsed;	//	Multim
bool	performance_timer;	//	Using '
int64	performance_timer_start;	//	Perfor
int64	performance_timer_elapsed;	//	Perfori
timer;		//	Struct

The next line of code is our speed table. The objects in the game will move at a different rate depending on the value of **adjust**. If **adjust** is 0 the objects will move one pixel at a time. If the value of **adjust** is 5, the objects will move 20 pixels at a time. So by increasing the value of **adjust** the speed of the objects will increase, making the game run faster on slow computers. The higher **adjust** is however, the choppier the game will play.

Basically **steps[]** is just a look-up table. If **adjust** was 3, we would look at the number stored at location 3 in **steps[]**. Location 0 holds the value 1, location 1 holds the value 2, location 2 holds the value 4, and location 3 hold the value 5. If **adjust** was 3, our objects would move 5 pixels at a time. Make sense?

int steps[6]={ 1, 2, 4, 5, 10, 20 };

Next we make room for two textures. We'll load a background scene, and a bitmap font texture. Then we set up a **base** variable so we can keep track of our font display list just like we did in the other font tutorials. Finally we declare WndProc().

GLuint texture[2]; GLuint base; LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

// Declara

// Steppi:

}

Now for the fun stuff :) The next section of code initializes our timer. It will check the computer to see if a performance counter is available (very accurate counter). If we don't have a performance counter the computer will use the multimedia timer. This code should be portable from what I'm told.

We start off by clearing all the timer variables to zero. This will set all the variables in our timer structure to zero. After that, we check to see if there is NOT a performance counter. The ! means NOT. If there is, the frequency will be stored in **timer.frequency**.

If there was no performance counter, the code in between the { }'s is run. The first line sets the variable **timer.performance_timer** to FALSE. This tells our program that there is no performance counter. The second line gets our starting multimedia timer value from timeGetTime(). We set the **timer.resolution** to 0.001f, and the **timer.frequency** to 1000. Because no time has elapsed yet, we make the elapsed time equal the start time.

```
void TimerInit(void)
{
         memset(&timer, 0, sizeof(timer));
                                                                                     // Clear (
         // Check To See If A Performance Counter Is Available
         // If One Is Available The Timer Frequency Will Be Updated
         if (!QueryPerformanceFrequency((LARGE_INTEGER *) &timer.frequency))
         {
                  // No Performace Counter Available
                  timer.performance_timer = FALSE;
timer.mm_timer_start = timeGetTime();
                                                                                     // Set Pe:
                                                                                     // Use ti
                  timer.resolution = 1.0f/1000.0f;
                                                                                     // Set Ou:
                                       = 1000;
                  timer.frequency
                  timer.mm_timer_elapsed
                                             = timer.mm_timer_start;
         }
```

If there is a performance counter, the following code is run instead. The first line grabs the current starting value of the performance counter, and stores it in **timer.performance_timer_start**. Then we set **timer.performance_timer** to TRUE so that our program knows there is a performance counter available. After that we calculate the timer resolution by using the frequency that we got when we checked for a performance counter in the code above. We divide 1 by the frequency to get the resolution. The last thing we do is make the elapsed time the same as the starting time.

Notice instead of sharing variables for the performance and multimedia timer start and elapsed variables, I've decided to make seperate variables. Either way it will work fine.

```
else
{
    // Performance Counter Is Available, Use It Instead Of The Multimedia Tim
    // Get The Current Time And Store It In performance_timer_start
    QueryPerformanceCounter((LARGE_INTEGER *) &timer.performance_timer_start)
    timer.performance_timer = TRUE;
    // Calculate The Timer Resolution Using The Timer Frequency
    timer.resolution = (float) (((double)1.0f)/((double)timer.frequ
    // Set The Elapsed Time To The Current Time
    timer.performance_timer_elapsed = timer.performance_timer_start;
}
```

The section of code above sets up the timer. The code below reads the timer and returns the amount of time that has passed in milliseconds.

The first thing we do is set up a 64 bit variable called **time**. We will use this variable to grab the current counter value. The next line checks to see if we have a performance counter. If we do, **timer.performance_timer** will be TRUE and the code right after will run.

The first line of code inside the { }'s grabs the counter value and stores it in the variable we created called **time**. The second line takes the time we just grabbed (**time** and subtracts the start time that we got when we initialized the timer. This way our timer should start out pretty close to zero. We then multiply the results by the resolution to find out how many seconds have passed. The last thing we do is multiply the result by 1000 to figure out how many milliseconds have passed. After the calculation is done, our results are sent back to the section of code that called this procedure. The results will be in floating point format for greater accuracy.

If we are not using the peformance counter, the code after the else statement will be run. It does pretty much the same thing. We grab the current time with timeGetTime() and subtract our starting counter value. We multiply it by our resolution and then multiply the result by 1000 to convert from seconds into milliseconds.

```
float TimerGetTime()
ł
         ___int64 time;
         if (timer.performance_timer)
         {
                  QueryPerformanceCounter((LARGE_INTEGER *) &time);
                                                                                    // Grab T
                  // Return The Current Time Minus The Start Time Multiplied By The Resolut.
                  return ( (float) ( time - timer.performance_timer_start) * timer.resoluti
         }
         else
         {
                  // Return The Current Time Minus The Start Time Multiplied By The Resolut.
                  return( (float) ( timeGetTime() - timer.mm_timer_start) * timer.resolutio:
         }
}
```

The following section of code resets the player to the top left corner of the screen, and gives the enemies a random starting point.

The top left of the screen is 0 on the x-axis and 0 on the y-axis. So by setting the **player.x** value to 0 we move the player to the far left side of the screen. By setting the **player.y** value to 0 we move our player to the top of the screen.

The fine positions have to be equal to the current player position, otherwise our player would move from whatever value it's at on the fine position to the top left of the screen. We don't want to player to move there, we want it to appear there, so we set the fine positions to 0 as well.

```
void ResetObjects(void)
{
    player.x=0;
    player.y=0;
    player.fx=0;
    player.fy=0;
```

}

Next we give the enemies a random starting location. The number of enemies displayed on the screen will be equal to the current (internal) **level** value multiplied by the current stage. Remember, the maximum value that **level** can equal is 3 and the maximum number of stages per level is 3. So we can have a total of 9 enemies.

To make sure we give all the viewable enemies a new position, we loop through all the visible enemies (**stage** times **level**). We set each enemies x position to 5 plus a random value from 0 to 5. (the maximum value rand can be is always the number you specify minus 1). So the enemy can appear on the grid, anywhere from 5 to 10. We then give the enemy a random value on the y axis from 0 to 10.

We don't want the enemy to move from it's old position to the new random position so we make sure the fine x (fx) and y (fy) values are equal to the actual x and y values multiplied by width and height of each tile on the screen. Each tile has a width of 60 and a height of 40.

The AUX_RGBImageRec code hasn't changed so I'm skipping over it. In LoadGLTextures() we will load in our two textures. First the font bitmap (**Font.bmp**) and then the background image (**Image.bmp**). We'll convert both the images into textures that we can use in our game. After we have built the textures we clean up by deleting the bitmap information. Nothing really new. If you've read the other tutorials you should have no problems understanding the code.

```
int LoadGLTextures()
ł
         int Status=FALSE;
                                                                                      // Status
                                                                                      // Create
         AUX_RGBImageRec *TextureImage[2];
         memset(TextureImage, 0, sizeof(void *)*2);
                                                                                      // Set The
                   ((TextureImage[0]=LoadBMP("Data/Font.bmp")) &&
         if
                   (TextureImage[1]=LoadBMP("Data/Image.bmp")))
                                                                                      // Load B
         {
                   Status=TRUE;
                   glGenTextures(2, &texture[0]);
                   for (loop1=0; loop1<2; loop1++)</pre>
                   {
                            glBindTexture(GL_TEXTURE_2D, texture[loop1]);
                            glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop1]->sizeX, Te
                            glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
                            glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                   }
                   for (loop1=0; loop1<2; loop1++)</pre>
                   {
                            if (TextureImage[loop1])
                                                                                      // If Tex
                            {
                                      if (TextureImage[loop1]->data)
```

```
{
    free(TextureImage[loop1]->data); // Free Ti
}
free(TextureImage[loop1]); // Free Ti
}
}
return Status;
```

}

The code below builds our font display list. I've already done a tutorial on bitmap texture fonts. All the code does is divides the **Font.bmp** image into 16 x 16 cells (256 characters). Each 16x16 cell will become a character. Because I've set the y-axis up so that positive goes down instead of up, it's necessary to subtract our y-axis values from 1.0f. Otherwise the letters will all be upside down :) If you don't understand what's going on, go back and read the bitmap texture font tutorial.

```
GLvoid BuildFont(GLvoid)
                                                                                    // Build (
{
         base=glGenLists(256);
         glBindTexture(GL_TEXTURE_2D, texture[0]);
                                                                                    // Select
         for (loop1=0; loop1<256; loop1++)</pre>
                                                                                    // Loop T
         {
                  float cx=float(loop1%16)/16.0f;
                  float cy=float(loop1/16)/16.0f;
                                                                                    // Start 1
                  glNewList(base+loop1,GL_COMPILE);
                            glBegin(GL_QUADS);
                                                                                    // Use A (
                                     glTexCoord2f(cx,1.0f-cy-0.0625f);
                                                                                    // Textur
                                     glVertex2d(0,16);
                                                                                    // Vertex
                                     glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f);
                                                                                    // Textur
                                     glVertex2i(16,16);
                                                                                    // Vertex
                                     glTexCoord2f(cx+0.0625f,1.0f-cy);
                                                                                    // Textur
                                     glVertex2i(16,0);
                                                                                    // Vertex
                                     glTexCoord2f(cx,1.0f-cy);
                                                                                    // Textur
                                     glVertex2i(0,0);
                                                                                    // Vertex
                            glEnd();
                                                                                    // Done B
                            glTranslated(15,0,0);
                  glEndList();
         }
}
```

ſ

It's a good idea to destroy the font display list when you're done with it, so I've added the following section of code. Again, nothing new.

```
GLvoid KillFont(GLvoid)
{
    glDeleteLists(base,256);
}
```

// Delete

The glPrint() code hasn't changed that much. The only difference from the tutorial on bitmap font textures is that I have added the ability to print the value of variables. The only reason I've written this section of code out is so that you can see the changes. The print statement will position the text at the **x** and **y** position that you specify. You can pick one of 2 character sets, and the value of variables will be written to the screen. This allows us to display the current **level** and **stage** on the screen.

Notice that I enable texture mapping, reset the view and then translate to the proper x / y position. Also notice that if character **set** 0 is selected, the font is enlarged one and half times width wise, and double it's original size up and down. I did this so that I could write the title of the game in big letters. After the text has been drawn, I disable texture mapping.

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...)
                                                                                   // Where '
ł
         char
                          text[256];
        va_list
                          ap;
         if (fmt == NULL)
                                                                                   // If The:
                 return;
                                                                                   // Parses
         va_start(ap, fmt);
            vsprintf(text, fmt, ap);
         va_end(ap);
         if (set>1)
         {
                  set=1;
         }
         glEnable(GL_TEXTURE_2D);
                                                                                   // Enable
         glLoadIdentity();
                                                                                   // Reset '
         glTranslated(x,y,0);
         glListBase(base-32+(128*set));
         if (set==0)
         {
                  glScalef(1.5f,2.0f,1.0f);
                                                                                   // Enlarg
         }
                                                                                   // Write '
         glCallLists(strlen(text),GL_UNSIGNED_BYTE, text);
                                                                                   // Disabl
         glDisable(GL_TEXTURE_2D);
}
```

The resize code is NEW :) Instead of using a perspective view I'm using an ortho view for this tutorial. That means that objects don't get smaller as they move away from the viewer. The z-axis is pretty much useless in this tutorial.

We start off by setting up the view port. We do this the same way we'd do it if we were setting up a perspective view. We make the viewport equal to the width of our window.

Then we select the projection matrix (thing movie projector, it information on how to display our image). and reset it.

Immediately after we reset the projection matrix, we set up our ortho view. I'll explain the command in detail:

The first parameter (0.0f) is the value that we want for the far left side of the screen. You wanted to know how to use actual pixel values, so instead of using a negative number for far left, I've set the value to 0. The second parameter is the value for the far right side of the screen. If our window is 640x480, the value stored in **width** will be 640. So the far right side of the screen effectively

becomes 640. Therefore our screen runs from 0 to 640 on the x-axis.

The third parameter (height) would normally be our negative y-axis value (bottom of the screen). But because we want exact pixels, we wont have a negative value. Instead we will make the bottom of the screen equal the **height** of our window. If our window is 640x480, **height** will be equal to 480. So the bottom of our screen will be 480. The fourth parameter would normally be the positive value for the top of our screen. We want the top of the screen to be 0 (good old fashioned screen coordinates) so we just set the fourth parameter to 0. This gives us from 0 to 480 on the y-axis.

The last two parameters are for the z-axis. We don't really care about the z-axis so we'll set the range from -1.0f to 1.0f. Just enough that we can see anything drawn at 0.0f on the z-axis.

After we've set up the ortho view, we select the modelview matrix (object information... location, etc) and reset it.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
                                                                                     // Resize
         if (height==0)
         {
                  height=1;
                                                                                     // Making
         }
         glViewport(0,0,width,height);
         glMatrixMode(GL_PROJECTION);
         glLoadIdentity();
                                                                                     // Reset '
         glOrtho(0.0f,width,height,0.0f,-1.0f,1.0f);
                                                                                     // Create
         glMatrixMode(GL_MODELVIEW);
                                                                                     // Select
         glLoadIdentity();
                                                                                     // Reset '
```

}

The init code has a few new commands. We start off by loading our textures. If they didn't load properly, the program will quit with an error message. After we have built the textures, we build our font set. I don't bother error checking but you can if you want.

After the font has been built, we set things up. We enable smooth shading, set our clear color to black and set depth clearing to 1.0f. After that is a new line of code.

glHint() tells OpenGL how to draw something. In this case we are telling OpenGL that we want line smoothing to be the best (nicest) that OpenGL can do. This is the command that enables antialiasing.

The last thing we do is enable blending and select the blend mode that makes anti-aliased lines possible. Blending is required if you want the lines to blend nicely with the background image. Disable blending if you want to see how crappy things look without it.

It's important to point out that antialiasing may not appear to be working. The objects in this game are quite small so you may not notice the antialaising right off the start. Look hard. Notice how the jaggie lines on the enemies smooth out when antialiasing is on. The player and hourglass should look better as well.

```
int InitGL(GLvoid)
{
    if (!LoadGLTextures())
    {
```

// All Se

```
return FALSE;
}
BuildFont();
glShadeModel(GL_SMOOTH); // Enable
glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
glClearDepth(1.0f);
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Type 0.
return TRUE;
}
```

Now for the drawing code. This is where the magic happens :)

We clear the screen (to black) along with the depth buffer. Then we select the font texture (texture [0]). We want the words "GRID CRAZY" to be a purple color so we set red and blue to full intensity, and we turn the green up half way. After we've selected the color, we call glPrint(). We position the words "GRID CRAZY" at 207 on the x axis (center on the screen) and 24 on the y-axis (up and down). We use our large font by selecting font **set** 0.

After we've drawn "GRID CRAZY" to the screen, we change the color to yellow (full red, full green). We write "Level:" and the variable **level2** to the screen. Remember that **level2** can be greater than 3. **level2** holds the level value that the player sees on the screen. %2i means that we don't want any more than 2 digits on the screen to represent the level. The i means the number is an integer number.

After we have written the level information to the screen, we write the stage information right under it using the same color.

```
int DrawGLScene(GLvoid)
```

ł

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear;
glBindTexture(GL_TEXTURE_2D, texture[0]);
glColor3f(1.0f,0.5f,1.0f); // Set Co
glPrint(207,24,0,"GRID CRAZY");
glColor3f(1.0f,1.0f,0.0f); // Set Co
glPrint(20,20,1,"Level:%2i",level2); // Write;
glPrint(20,40,1,"Stage:%2i",stage); // Write;
```

Now we check to see if the game is over. If the game is over, the variable **gameover** will be TRUE. If the game is over, we use glColor3ub(r,g,b) to select a random color. Notice we are using 3ub instead of 3f. By using 3ub we can use integer values from 0 to 255 to set our colors. Plus it's easier to get a random value from 0 to 255 than it is to get a random value from 0.0f to 1.0f.

Once a random color has been selected, we write the words "GAME OVER" to the right of the game title. Right under "GAME OVER" we write "PRESS SPACE". This gives the player a visual message letting them know that they have died and to press the spacebar to restart the game.

```
if (gameover)
{
    glColor3ub(rand()%255,rand()%255,rand()%255); // Pick A
    glPrint(472,20,1,"GAME OVER");
    glPrint(456,40,1,"PRESS SPACE"); // Write 1
}
```

If the player still has lives left, we draw animated images of the players character to the right of the game title. To do this we create a loop that goes from 0 to the current number of **lives** the player has left minus one. I subtract one, because the current life is the image you control.

Inside the loop, we reset the view. After the view has been reset, we translate to the 490 pixels to the right plus the value of **loop1** times 40.0f. This draws each of the animated player lives 40 pixels apart from eachother. The first animated image will be drawn at 490+(0*40) (= 490), the second animated image will be drawn at 490+(1*40) (= 530), etc.

After we have moved to the spot we want to draw the animated image, we rotate counterclockwise depending on the value stored in **player.spin**. This causes the animated life images to spin the opposite way that your active player is spinning.

We then select green as our color, and start drawing the image. Drawing lines is alot like drawing a quad or a polygon. You start off with glBegin(GL_LINES), telling OpenGL we want to draw a line. Lines have 2 vertices. We use glVertex2d to set our first point. glVertex2d doesn't require a z value, which is nice considering we don't care about the z value. The first point is drawn 5 pixels to the left of the current x location and 5 pixels up from the current y location. Giving us a top left point. The second point of our first line is drawn 5 pixels to the right of our current x location, and 5 pixels down, giving us a bottom right point. This draws a line from the top left to the bottom right. Our second line is drawn from the top right to the bottom left. This draws a green X on the screen.

After we have drawn the green X, we rotate counterclockwise (on the z axis) even more, but this time at half the speed. We then select a darker shade of green (0.75f) and draw another x, but we use 7 instead of 5 this time. This draws a bigger / darker x on top of the first green X. Because the darker X spins slower though, it will look as if the bright X has a spinning set of feelers (grin) on top of it.

```
for (loop1=0; loop1<lives-1; loop1++)</pre>
{
         glLoadIdentity();
                                                                            // Reset '
         glTranslatef(490+(loop1*40.0f),40.0f,0.0f);
                                                                            // Move T
         glRotatef(-player.spin,0.0f,0.0f,1.0f);
         glColor3f(0.0f,1.0f,0.0f);
                                                                            // Set Pla
         glBegin(GL_LINES);
                                                                            // Start 1
                  glVertex2d(-5,-5);
                                                                            // Top Le:
                                                                            // Bottom
                  glVertex2d( 5, 5);
                  glVertex2d( 5,-5);
                                                                            // Top Rig
                  glVertex2d(-5, 5);
                                                                            // Bottom
         glEnd();
                                                                            // Done D:
         glRotatef(-player.spin*0.5f,0.0f,0.0f,1.0f);
                                                                            // Rotate
         glColor3f(0.0f,0.75f,0.0f);
                                                                            // Set Pla
         glBegin(GL_LINES);
                                                                            // Start 1
                  glVertex2d(-7, 0);
                                                                            // Left C
                  glVertex2d( 7, 0);
                                                                            // Right (
                  glVertex2d( 0,-7);
                                                                            // Top Cei
                  glVertex2d( 0, 7);
                                                                            // Bottom
         glEnd();
                                                                            // Done D:
}
```

Now we're going to draw the grid. We set the variable **filled** to TRUE. This tells our program that the grid has been completely filled in (you'll see why we do this in a second).

Right after that we set the line width to 2.0f. This makes the lines thicker, making the grid look more defined.

Then we disable anti-aliasing. The reason we disable anti-aliasing is because although it's a great feature, it eats CPU's for breakfast. Unless you have a killer graphics card, you'll notice a huge slow down if you leave anti-aliasing on. Go ahead and try if you want :)

The view is reset, and we start two loops. **loop1** will travel from left to right. **loop2** will travel from top to bottom.

We set the line color to blue, then we check to see if the horizontal line that we are about to draw has been traced over. If it has we set the color to white. The value of **hline[loop1][loop2]** will be TRUE if the line has been traced over, and FALSE if it hasn't.

After we have set the color to blue or white, we draw the line. The first thing to do is make sure we haven't gone to far to the right. We don't want to draw any lines or check to see if the line has been filled in when **loop1** is greater than 9.

Once we are sure **loop1** is in the valid range we check to see if the horizontal line hasn't been filled in. If it hasn't, **filled** is set to FALSE, letting our OpenGL program know that there is at least one line that hasn't been filled in.

The line is then drawn. We draw our first horizontal (left to right) line starting at 20+(0*60) (= 20). This line is drawn all the way to 80+(0*60) (= 80). Notice the line is drawn to the right. That is why we don't want to draw 11 (0-10) lines. because the last line would start at the far right of the screen and end 80 pixels off the screen.

```
filled=TRUE;
                                                                             // Set Li
glLineWidth(2.0f);
glDisable(GL_LINE_SMOOTH);
                                                                              // Disabl
glLoadIdentity();
                                                                              // Reset '
                                                                              // Loop F:
for (loop1=0; loop1<11; loop1++)</pre>
{
         for (loop2=0; loop2<11; loop2++)</pre>
                                                                             // Loop F:
          {
                   glColor3f(0.0f,0.5f,1.0f);
                                                                             // Set Li
                   if (hline[loop1][loop2])
                                                                             // Has The
                   {
                             glColor3f(1.0f,1.0f,1.0f);
                                                                             // If So,
                   }
                   if (loop1<10)
                   {
                             if (!hline[loop1][loop2])
                                                                             // If A H
                             {
                                      filled=FALSE;
                             }
                             glBegin(GL_LINES);
                                                                             // Start 1
                                      glVertex2d(20+(loop1*60),70+(loop2*40));
                                      glVertex2d(80+(loop1*60),70+(loop2*40));
                                                                              // Done D:
                             glEnd();
                   }
```

The code below does the same thing, but it checks to make sure the line isn't being drawn too far down the screen instead of too far right. This code is responsible for drawing vertical lines.

```
glColor3f(0.0f,0.5f,1.0f);
                                                        // Set Li
if (vline[loop1][loop2])
                                                        // Has Th
{
         glColor3f(1.0f,1.0f,1.0f);
                                                        // If So,
}
if (loop2<10)
{
         if (!vline[loop1][loop2])
                                                        // If A V
         {
                  filled=FALSE;
         }
         glBegin(GL_LINES);
                                                        // Start 1
                  glVertex2d(20+(loop1*60),70+(loop2*40));
                  glVertex2d(20+(loop1*60),110+(loop2*40));
         glEnd();
                                                        // Done D:
}
```

Now we check to see if 4 sides of a box are traced. Each box on the screen is 1/10th of a full screen picture. Because each box is piece of a larger texture, the first thing we need to do is enable texture mapping. We don't want the texture to be tinted red, green or blue so we set the color to bright white. After the color is set to white we select our grid texture (**texture[1]**).

The next thing we do is check to see if we are checking a box that exists on the screen. Remember that our loop draws the 11 lines right and left and 11 lines up and down. But we dont have 11 boxes. We have 10 boxes. So we have to make sure we don't check the 11th position. We do this by making sure both **loop1** and **loop2** is less than 10. That's 10 boxes from 0 - 9.

After we have made sure that we are in bounds we can start checking the borders. **hline[loop1]** [loop2] is the top of a box. **hline[loop1][loop2+1]** is the bottom of a box. **vline[loop1][loop2]** is the left side of a box and **vline[loop1+1][loop2]** is the right side of a box. Hopefully I can clear things up with a diagram:

loop1,	100p2	hline	loop1+	1, 100p2
				Const.
v				v
1				1
1				4
n				n
e				e
loop1,	100p2+1	hline	100p1+1,	100p2+1

All horizontal lines are assumed to run from **loop1** to **loop1**+1. As you can see, the first horizontal line is runs along **loop2**. The second horizontal line runs along **loop2**+1. Vertical lines are assumed to run from **loop2** to **loop2**+1. The first vertical line runs along **loop1** and the second vertical line runs along **loop1**+1

When **loop1** is increased, the right side of our old box becomes the left side of the new box. When **loop2** is increased, the bottom of the old box becomes the top of the new box.

If all 4 borders are TRUE (meaning we've passed over them all) we can texture map the box. We do this the same way we broke the font texture into seperate letters. We divide both **loop1** and **loop2** by 10 because we want to map the texture across 10 boxes from left to right and 10 boxes up and down. Texture coordinates run from 0.0f to 1.0f and 1/10th of 1.0f is 0.1f.

So to get the top right side of our box we divide the loop values by 10 and add 0.1f to the x texture coordinate. To get the top left side of the box we divide our loop values by 10. To get the bottom left side of the box we divide our loop values by 10 and add 0.1f to the y texture coordinate. Finally to get the bottom right texture coordinate we divide the loop values by 10 and add 0.1f to both the x and y texture coordinates.

Quick examples:

loop1=0 and loop2=0

- Right X Texture Coordinate = **loop1**/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f
- Left X Texture Coordinate = **loop1**/10 = 0/10 = 0.0f
- Top Y Texture Coordinate = **loop2**/10 = 0/10 = 0.0f;
- Bottom Y Texture Coordinate = **loop2**/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f;

loop1=1 and loop2=1

- Right X Texture Coordinate = **loop1**/10+0.1f = 1/10+0.1f = 0.1f+0.1f = 0.2f
- Left X Texture Coordinate = **loop1**/10 = 1/10 = 0.1f
- Top Y Texture Coordinate = **loop2**/10 = 1/10 = 0.1f;
- Bottom Y Texture Coordinate = **loop2**/10+0.1f = 1/10+0.1f = 0.1f+0.1f = 0.2f;

Hopefully that all makes sense. If **loop1** and **loop2** were equal to 9 we would end up with the values 0.9f and 1.0f. So as you can see our texture coordinates mapped across the 10 boxes run from 0.0f at the lowest and 1.0f at the highest. Mapping the entire texture to the screen. After we've mapped a section of the texture to the screen, we disable texture mapping. Once we've drawn all the lines and filled in all the boxes, we set the line width to 1.0f.



The code below checks to see if anti is TRUE. If it is, we enable line smoothing (anti-aliasing).

```
if (anti) // Is Ant
{
    glEnable(GL_LINE_SMOOTH); // If So,
}
```

}

To make the game a little easier I've added a special item. The item is an hourglass. When you touch the hourglass, the enemies are frozen for a specific amount of time. The following section of code is resposible for drawing the hourglass.

For the hourglass we use **x** and **y** to position the timer, but unlike our player and enemies we don't use **fx** and **fy** for fine positioning. Instead we'll use **fx** to keep track of whether or not the timer is being displayed. **fx** will equal 0 if the timer is not visible. 1 if it is visible, and 2 if the player has touched the timer. **fy** will be used as a counter to keep track of how long the timer should be visible or invisible.

So we start off by checking to see if the timer is visible. If not, we skip over the code without drawing the timer. If the timer is visible, we reset the modelview matrix, and position the timer. Because our first grid point from left to right starts at 20, we will add **hourglass.x** times 60 to 20. We multiply **hourglass.x** by 60 because the points on our grid from left to right are spaced 60 pixels apart. We then position the hourglass on the y axis. We add **hourglass.y** times 40 to 70.0f because we want to start drawing 70 pixels down from the top of the screen. Each point on our grid from top to bottom is spaced 40 pixels apart.

After we have positioned the hourglass, we can rotate it on the z-axis. **hourglass.spin** is used to keep track of the rotation, the same way player.spin keeps track of the player rotation. Before we start to draw the hourglass we select a random color.

glBegin(GL_LINES) tells OpenGL we want to draw using lines. We start off by moving left and up 5 pixels from our current location. This gives us the top left point of our hourglass. OpenGL will start drawing the line from this location. The end of the line will be 5 pixels right and down from our original location. This gives us a line running from the top left to the bottom right. Immediately after that we draw a second line running from the top right to the bottom left. This gives us an 'X'. We finish off by connecting the bottom two points together, and then the top two points to create an hourglass type object :)

glBegin(GL_LINES);	// Start 1
glVertex2d(-5,-5);	// Top Le
glVertex2d(5, 5);	// Bottom
glVertex2d(5,-5);	// Top Rig
glVertex2d(-5, 5);	// Bottom
glVertex2d(-5, 5);	// Bottom
glVertex2d(5, 5);	// Bottom
glVertex2d(-5,-5);	// Top Le
glVertex2d(5,-5);	// Top Rig
glEnd();	// Done D:

Now we draw our player. We reset the modelview matrix, and position the player on the screen. Notice we position the player using **fx** and **fy**. We want the player to move smoothly so we use fine positioning. After positioning the player, we rotate the player on it's z-axis using **player.spin**. We set the color to light green and begin drawing. Just like the code we used to draw the hourglass, we draw an 'X'. Starting at the top left to the bottom right, then from the top right to the bottom left.

```
// Reset '
glLoadIdentity();
glTranslatef(player.fx+20.0f,player.fy+70.0f,0.0f);
                                                                          // Move T
glRotatef(player.spin,0.0f,0.0f,1.0f);
glColor3f(0.0f,1.0f,0.0f);
                                                                          // Set Pla
glBegin(GL_LINES);
                                                                          // Start 1
         glVertex2d(-5,-5);
                                                                          // Top Le:
         glVertex2d( 5, 5);
                                                                          // Bottom
         glVertex2d( 5,-5);
                                                                          // Top Rig
                                                                          // Bottom
         glVertex2d(-5, 5);
                                                                           // Done D:
glEnd();
```

Drawing low detail objects with lines can be a little frustrating. I didn't want the player to look boring so I added the next section of code to create a larger and quicker spinning blade on top of the player that we drew above. We rotate on the z-axis by **player.spin** times 0.5f. Because we are rotating again, it will appear as if this piece of the player is moving a little quicker than the first piece of the player.

After doing the new rotation, we set the color to a darker shade of green. So that it actually looks like the player is made up of different colors / pieces. We then draw a large '+' on top of the first piece of the player. It's larger because we're using -7 and +7 instead of -5 and +5. Also notice that instead of drawing from one corner to another, I'm drawing this piece of the player from left to right and top to bottom.

glRotatef(player.spin*0.5f,0.0f,0.0f,1.0f);	// Rotate
glColor3f(0.0f,0.75f,0.0f);	
glBegin(GL_LINES);	
glVertex2d(-7, 0);	// Left C
glVertex2d(7, 0);	// Right (
glVertex2d(0,-7);	// Top Cei
glVertex2d(0, 7);	// Bottom
glEnd();	// Done D:

All we have to do now is draw the enemies, and we're done drawing :) We start off by creating a loop that will loop through all the enemies visible on the current level. We calculate how many enemies to draw by multiplying our current game **stage** by the games internal **level**. Remember that each level has 3 stages, and the maximum value of the internal level is 3. So we can have a maximum of 9 enemies.

Inside the loop we reset the modelview matrix, and position the current enemy (**enemy[loop1]**). We position the enemy using it's fine x and y values (**fx** and **fy**). After positioning the current enemy we set the color to pink and start drawing.

The first line will run from 0, -7 (7 pixels up from the starting location) to -7,0 (7 pixels left of the starting location). The second line runs from -7,0 to 0,7 (7 pixels down from the starting location). The third line runs from 0,7 to 7,0 (7 pixels to the right of our starting location), and the last line runs from 7,0 back to the beginning of the first line (7 pixels up from the starting location). This creates a non spinning pink diamond on the screen.

```
for (loop1=0; loop1<(stage*level); loop1++)</pre>
                                                                             // Loop To
{
         glLoadIdentity();
                                                                            // Reset '
         glTranslatef(enemy[loop1].fx+20.0f,enemy[loop1].fy+70.0f,0.0f);
         glColor3f(1.0f,0.5f,0.5f);
                                                                            // Make E
         glBegin(GL_LINES);
                                                                            // Start 1
                  glVertex2d( 0,-7);
                                                                            // Top Po.
                  glVertex2d(-7, 0);
                                                                            // Left P
                  glVertex2d(-7, 0);
                                                                            // Left P
                  glVertex2d( 0, 7);
                                                                            // Bottom
                  glVertex2d( 0, 7);
                                                                            // Bottom
                  glVertex2d( 7, 0);
                                                                            // Right :
                  glVertex2d( 7, 0);
                                                                            // Right :
                   glVertex2d( 0,-7);
                                                                            // Top Po.
                                                                             // Done D:
         glEnd();
```

We don't want the enemy to look boring either so we'll add a dark red spinning blade ('X') on top of the diamond that we just drew. We rotate on the z-axis by **enemy[loop1].spin**, and then draw the 'X'. We start at the top left and draw a line to the bottom right. Then we draw a second line from the top right to the bottom left. The two lines cross eachother creating an 'X' (or blade ... grin).

glRotatef(enemy[loop1].spin,0.0f,0.0f,1.0f);	// Rotate
glColor3f(1.0f,0.0f);	// Make E
glBegin(GL_LINES);	// Start 1
glVertex2d(-7,-7);	// Top Le
glVertex2d(7, 7);	// Bottom
glVertex2d(-7, 7);	// Bottom
glVertex2d(7,-7);	// Top Rig
glEnd();	// Done D:
}	
return TRUE;	
}	

I added the KillFont() command to the end of KillGLWindow(). This makes sure the font display list is destroyed when the window is destroyed.

```
GLvoid KillGLWindow(GLvoid)
                                                                                     // Proper
{
         if (fullscreen)
         {
                  ChangeDisplaySettings(NULL,0);
                   ShowCursor(TRUE);
                                                                                     // Show M
         }
         if (hRC)
                                                                                     // Do We 1
         {
                   if (!wglMakeCurrent(NULL,NULL))
                   {
                            MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR",
                   }
                   if (!wglDeleteContext(hRC))
                                                                                     // Are We
                   {
                            MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN ER
```

}

```
}
         hRC=NULL;
                                                                            // Set RC
}
if (hDC && !ReleaseDC(hWnd,hDC))
                                                                            // Are We
{
         MessageBox(NULL, "Release Device Context Failed.", "SHUTDOWN ERROR", MB_OK
         hDC=NULL;
                                                                            // Set DC
}
if (hWnd && !DestroyWindow(hWnd))
                                                                            // Are We
{
         MessageBox(NULL, "Could Not Release hWnd.", "SHUTDOWN ERROR", MB_OK | MB_ICO
         hWnd=NULL;
}
if (!UnregisterClass("OpenGL", hInstance))
                                                                            // Are We
{
         MessageBox(NULL, "Could Not Unregister Class.", "SHUTDOWN ERROR", MB_OK | MB.
         hInstance=NULL;
}
KillFont();
```

The CreateGLWindow() and WndProc() code hasn't changed so search until you find the following section of code.

int WINAPI WinMain(HINSTANCE HI HINSTANCE HE	Instance, PrevInstance,	// Instan // Previo
	LPSTR	lpCmdLine,	
	int	nCmdShow)	// Window
{			
MSG msg;			
BOOL done=F	ALSE;		
// Ask The User	Which Screen	Mode They Prefer	
if (MessageBox()	NULL, "Would Yo	u Like To Run In Fullscreen Mode	e?", "Start FullScreen'
{			
fullsc	reen=FALSE;		// Window
}			

This section of code hasn't changed that much. I changed the window title to read "NeHe's Line Tutorial", and I added the ResetObjects() command. This sets the player to the top left point of the grid, and gives the enemies random starting locations. The enemies will always start off at least 5 tiles away from you.

```
if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen)) // Create
{
    return 0; // Quit I:
}
ResetObjects();
while(!done)
{
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is The:
    {
}
```

Now to make the timing code work. Notice before we draw our scene we grab the time, and store it in a floating point variable called **start**. We then draw the scene and swap buffers.

Immediately after we swap the buffers we create a delay. We do this by checking to see if the current value of the timer (**TimerGetTime(**)) is less than our starting value plus the game stepping speed times 2. If the current timer value is less than the value we want, we endlessly loop until the current timer value is equal to or greater than the value we want. This slows down REALLY fast systems.

Because we use the stepping speed (set by the value of **adjust**) the program will always run the same speed. For example, if our stepping speed was 1 we would wait until the timer was greater than or equal to 2 (1*2). But if we increased the stepping speed to 2 (causing the player to move twice as many pixels at a time), the delay is increased to 4 (2*2). So even though we are moving twice as fast, the delay is twice as long, so the game still runs the same speed :)

One thing alot of people like to do is take the current time, and subtract the old time to find out how much time has passed. Then they move objects a certain distance based on the amount of time that has passed. Unfortunately I can't do that in this program because the fine movement has to be exact so that the player can line up with the lines on the grid. If the current fine x position was 59 and the computer decided the player needed to move two pixels, the player would never line up with the vertical line at position 60 on the grid.

The following code hasn't really changed. I changed the title of the window to read "NeHe's Line Tutorial".

if (keys[VK_F1])	// Is F1 1
{	
keys[VK_F1]=FALSE;	// If So 1
KillGLWindow();	

// Have W

}

This section of code checks to see if the A key is being pressed and not held. If 'A' is being pressed, **ap** becomes TRUE (telling our program that A is being held down), and **anti** is toggled from TRUE to FALSE or FALSE to TRUE. Remember that **anti** is checked in the drawing code to see if antialiasing is turned on or off.

If the 'A' key has been released (is FALSE) then **ap** is set to FALSE telling the program that the key is no longer being held down.

Now to move the enemies. I wanted to keep this section of code really simple. There is very little logic. Basically, the enemies check to see where you are and they move in that direction. Because I'm checking the actual **x** and **y** position of the players and no the fine values, the players seem to have a little more intelligence. They may see that you are way at the top of the screen. But by the time they're fine value actually gets to the top of the screen, you could already be in a different location. This causes them to sometimes move past you, before they realize you are no longer where they thought you were. May sound like they're really dumb, but because they sometimes move past you, you might find yourself being boxed in from all directions.

We start off by checking to make sure the game isn't over, and that the window (if in windowed mode) is still active. By checking **active** the enemies wont move if the screen is minimized. This gives you a convenient pause feature when you need to take a break :)

After we've made sure the enemies should be moving, we create a loop. The loop will loop through all the visible enemies. Again we calculate how many enemies should be on the screen by multiplying the current **stage** by the current internal **level**.

Now we move the current enemy (**enemy[loop1]**). We start off by checking to see if the enemy's **x** position is less than the players **x** position and we make sure that the enemy's fine y position lines up with a horizontal line. We can't move the enemy left and right if it's not on a horizontal line. If we did, the enemy would cut right through the middle of the boxes, making the game even more difficult :)

If the enemy \mathbf{x} position is less than the player \mathbf{x} position, and the enemy's fine \mathbf{y} position is lined up with a horizontal line, we move the enemy \mathbf{x} position one block closer to the current player position.

We also do this to move the enemy left, down and up. When moving up and down, we need to make sure the enemy's fine \mathbf{x} position lines up with a vertical line. We don't want the enemy cutting through the top or bottom of a box.

Note: changing the enemies \mathbf{x} and \mathbf{y} positions doesn't move the enemy on the screen. Remember that when we drew the enemies we used the fine positions to place the enemies on the screen. Changing the \mathbf{x} and \mathbf{y} positions just tells our program where we WANT the enemies to move.

if ((enemy[loop1].x<player.x) && (enemy[loop1]</pre> { enemy[loop1].x++; // Move T } if ((enemy[loop1].x>player.x) && (enemy[loop1] { enemy[loop1].x--; // Move T } if ((enemy[loop1].y<player.y) && (enemy[loop1])</pre> { enemy[loop1].y++; // Move T } if ((enemy[loop1].y>player.y) && (enemy[loop1] { enemy[loop1].y--; // Move T }

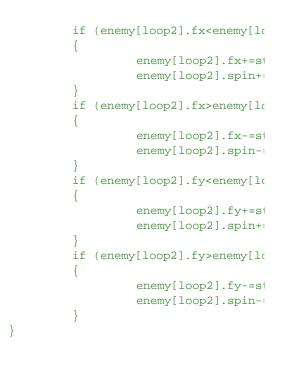
This code does the actual moving. We check to see if the variable **delay** is greater than 3 minus the current internal level. That way if our current level is 1 the program will loop through 2 (3-1) times before the enemies actually move. On level 3 (the highest value that **level** can be) the enemies will move the same speed as the player (no delays). We also make sure that **hourglass.fx** isn't the same as 2. Remember, if **hourglass.fx** is equal to 2, that means the player has touched the hourglass. Meaning the enemies shouldn't be moving.

If **delay** is greater than 3-**level** and the player hasn't touched the hourglass, we move the enemies by adjusting the enemy fine positions (**fx** and **fy**). The first thing we do is set **delay** back to 0 so that we can start the delay counter again. Then we set up a loop that loops through all the visible enemies (**stage** times **level**).

To move the enemies we check to see if the current enemy (**enemy[loop2]**) needs to move in a specific direction to move towards the enemy **x** and **y** position we want. In the first line below we check to see if the enemy fine position on the x-axis is less than the desired **x** position times 60. (remember each grid crossing is 60 pixels apart from left to right). If the fine **x** position is less than the enemy **x** position times 60 we move the enemy to the right by **steps[adjust]** (the speed our game is set to play at based on the value of **adjust**). We also rotate the enemy clockwise to make it look like it's rolling to the right. We do this by increasing **enemy[loop2].spin** by **steps[adjust]** (the current game speed based on **adjust**).

We then check to see if the enemy fx value is greater than the enemy x position times 60 and if so, we move the enemy left and spin the enemy left.

We do the same when moving the enemy up and down. If the enemy **y** position is less than the enemy **fy** position times 40 (40 pixels between grid points up and down) we increase the enemy **fy** position, and rotate the enemy to make it look like it's rolling downwards. Lastly if the enemy **y** position is greater than the enemy **fy** position times 40 we decrease the value of **fy** to move the enemy upward. Again, the enemy spins to make it look like it's rolling upward.



After moving the enemies we check to see if any of them have hit the player. We want accuracy so we compare the enemy fine positions with the player fine positions. If the enemy fx position equals the player fx position and the enemy fy position equals the player fy position the player is DEAD :)

}

If the player is dead, we decrease **lives**. Then we check to make sure the player isn't out of lives by checking to see if **lives** equals 0. If **lives** does equal zero, we set **gameover** to TRUE.

We then reset our objects by calling ResetObjects(), and play the death sound.

Sound is new in this tutorial. I've decided to use the most basic sound routine available... PlaySound(). PlaySound() takes three parameters. First we give it the name of the file we want to play. In this case we want it to play the Die .WAV file in the Data directory. The second parameter can be ignored. We'll set it to NULL. The third parameter is the flag for playing the sound. The two most common flags are: SND_SYNC which stops everything else until the sound is done playing, and SND_ASYNC, which plays the sound, but doesn't stop the program from running. We want a little delay after the player dies so we use SND_SYNC. Pretty easy! The one thing I forgot to mention at the beginning of the program: In order to use PlaySound(), you have to include the WINMM.LIB file under PROJECT / SETTINGS / LINK in Visual C++. Winmm.lib is the Windows Multimedia Library.

Now we can move the player. In the first line of code below we check to see if the right arrow is being pressed, **player.x** is less than 10 (don't want to go off the grid), that **player.fx** equals **player.x** times 60 (lined up with a grid crossing on the x-axis, and that **player.fy** equals **player.y** times 40 (player is lined up with a grid crossing on the y-axis).

}

If we didn't make sure the player was at a crossing, and we allowed the player to move anyways, the player would cut right through the middle of boxes, just like the enemies would have done if we didn't make sure they were lined up with a vertical or horizontal line. Checking this also makes sure the player is done moving before we move to a new location.

If the player is at a grid crossing (where a vertical and horizontal lines meet) and he's not to far right, we mark the current horizontal line that we are on as being traced over. We then increase the **player.x** value by one, causing the new player position to be one box to the right.

We do the same thing while moving left, down and up. When moving left, we make sure the player wont be going off the left side of the grid. When moving down we make sure the player wont be leaving the bottom of the grid, and when moving up we make sure the player doesn't go off the top of the grid.

When moving left and right we make the horizontal line (**hline[][]**) under us TRUE meaning it's been traced. When moving up and down we make the vertical line (**vline[][]**) under us TRUE meaning it has been traced.

We increase / decrease the player fine **fx** and **fy** variables the same way we increase / decreased the enemy fine **fx** and **fy** variables.

If the player **fx** value is less than the player **x** value times 60 we increase the player **fx** position by the step speed our game is running at based on the value of **adjust**.

If the player **fx** value is greater than the player **x** value times 60 we decrease the player **fx** position by the step speed our game is running at based on the value of **adjust**.

If the player **fy** value is less than the player **y** value times 40 we increase the player **fy** position by the step speed our game is running at based on the value of **adjust**.

If the player **fy** value is greater than the player **y** value times 40 we decrease the player **fy** position by the step speed our game is running at based on the value of **adjust**.

if	(player.fx <player.x*60)< th=""><th>// 1</th><th>Is Fin</th></player.x*60)<>	// 1	Is Fin
1	<pre>player.fx+=steps[adjust];</pre>	// 1	If So,
ر if	(player.fx>player.x*60)	// 1	Is Fin
1	<pre>player.fx-=steps[adjust];</pre>	// 1	If So,
ر if	(player.fy <player.y*40)< td=""><td>// 1</td><td>Is Fin</td></player.y*40)<>	// 1	Is Fin
1	<pre>player.fy+=steps[adjust];</pre>	// 1	If So,
} if	(player.fy>player.y*40)	// 1	ls Fin
1	<pre>player.fy-=steps[adjust];</pre>	// 1	If So,
}			

If the game is over the following bit of code will run. We check to see if the spacebar is being pressed. If it is we set **gameover** to FALSE (starting the game over). We set **filled** to TRUE. This causes the game to think we've finished a stage, causing the player to be reset, along with the enemies.

}

We set the starting level to 1, along with the actual displayed level (**level2**). We set **stage** to 0. The reason we do this is because after the computer sees that the grid has been filled in, it will think you finished a stage, and will increase **stage** by 1. Because we set **stage** to 0, when the stage increases it will become 1 (exactly what we want). Lastly we set **lives** back to 5.

```
else
{
    if (keys[' '])
    {
        gameover=FALSE;
        filled=TRUE;
        level=1;
    }
}
```

// Starti

}

```
level2=1;
stage=0;
lives=5;
}
```

// Display
// Game S
// Lives

The code below checks to see if the **filled** flag is TRUE (meaning the grid has been filled in). **filled** can be set to TRUE one of two ways. Either the grid is filled in completely and **filled** becomes TRUE or the game has ended but the spacebar was pressed to restart it (code above).

If **filled** is TRUE, the first thing we do is play the cool level complete tune. I've already explained how PlaySound() works. This time we'll be playing the Complete .WAV file in the DATA directory. Again, we use SND_SYNC so that there is a delay before the game starts on the next stage.

After the sound has played, we increase **stage** by one, and check to make sure **stage** isn't greater than 3. If **stage** is greater than 3 we set **stage** to 1, and increase the internal level and visible level by one.

If the internal level is greater than 3 we set the internal leve (**level**) to 3, and increase **lives** by 1. If you're amazing enough to get past level 3 you deserve a free life :). After increasing **lives** we check to make sure the player doesn't have more than 5 lives. If **lives** is greater than 5 we set **lives** back to 5.

```
if (filled)
{
         PlaySound("Data/Complete.wav", NULL, SND_SYNC);
                                                       // Increa
         stage++;
         if (stage>3)
         {
                  stage=1;
                                                       // If So,
                                                       // Increa
                  level++;
                  level2++;
                                                       // Increa
                  if (level>3)
                  {
                           level=3;
                                                       // If So,
                                                       // Give T
                           lives++;
                           if (lives>5)
                            {
                                                       // If So,
                                     lives=5;
                            }
                  }
         }
```

We then reset all the objects (such as the player and enemies). This places the player back at the top left corner of the grid, and gives the enemies random locations on the grid.

We create two loops (**loop1** and **loop2**) to loop through the grid. We set all the vertical and horizontal lines to FALSE. If we didn't do this, the next stage would start, and the game would think the grid was still filled in.

Notice the routine we use to clear the grid is similar to the routine we use to draw the grid. We have to make sure the lines are not being drawn to far right or down. That's why we check to make sure that **loop1** is less than 10 before we reset the horizontal lines, and we check to make sure that **loop2** is less than 10 before we reset the vertical lines.

Now we check to see if the player has hit the hourglass. If the fine player **fx** value is equal to the hourglass **x** value times 60 and the fine player **fy** value is equal to the hourglass **y** value times 40 AND **hourglass.fx** is equal to 1 (meaning the hourglass is displayed on the screen), the code below runs.

The first line of code is PlaySound("Data/freeze.wav",NULL, SND_ASYNC | SND_LOOP). This line plays the freeze .WAV file in the DATA directory. Notice we are using SND_ASYNC this time. We want the freeze sound to play without the game stopping. SND_LOOP keeps the sound playing endlessly until we tell it to stop playing, or until another sound is played.

After we have started the sound playing, we set **hourglass.fx** to 2. When **hourglass.fx** equals 2 the hourglass will no longer be drawn, the enemies will stop moving, and the sound will loop endlessly.

We also set **hourglass.fy** to 0. **hourglass.fy** is a counter. When it hits a certain value, the value of **hourglass.fx** will change.

This bit of code increases the player spin value by half the speed that the game runs at. If **player.spin** is greater than 360.0f we subtract 360.0f from **player.spin**. Keeps the value of **player.spin** from getting to high.

The code below decreases the hourglass spin value by 1/4 the speed that the game is running at. If **hourglass.spin** is less than 0.0f we add 360.0f. We don't want **hourglass.spin** to become a negative number.

The first line below increased the hourglass counter that I was talking about. **hourglass.fy** is increased by the game speed (game speed is the steps value based on the value of **adjust**).

The second line checks to see if **hourglass.fx** is equal to 0 (non visible) and the hourglass counter (**hourglass.fy**) is greater than 6000 divided by the current internal level (**level**).

If the **fx** value is 0 and the counter is greater than 6000 divided by the internal level we play the hourglass .WAV file in the DATA directory. We don't want the action to stop so we use SND_ASYNC. We won't loop the sound this time though, so once the sound has played, it wont play again.

After we've played the sound we give the hourglass a random value on the x-axis. We add one to the random value so that the hourglass doesn't appear at the players starting position at the top left of the grid. We also give the hourglass a random value on the y-axis. We set **hourglass.fx** to 1 this makes the hourglass appear on the screen at it's new location. We also set **hourglass.fy** back to zero so it can start counting again.

This causes the hourglass to appear on the screen after a fixed amount of time.

If **hourglass.fx** is equal to zero and **hourglass.fy** is greater than 6000 divided by the current internal level (**level**) we set **hourglass.fx** back to 0, causing the hourglass to disappear. We also set **hourglass.fy** to 0 so it can start counting once again.

This causes the hourglass to disappear if you don't get it after a certain amount of time.

// Spin T

// Is The

Now we check to see if the 'freeze enemy' timer has run out after the player has touched the hourglass.

if **hourglass.fx** equal 2 and **hourglass.fy** is greater than 500 plus 500 times the current internal level we kill the timer sound that we started playing endlessly. We kill the sound with the command PlaySound(NULL, NULL, 0). We set **hourglass.fx** back to 0, and set **hourglass.fy** to 0. Setting **fx** and **fy** to 0 starts the hourglass cycle from the beginning. **fy** will have to hit 6000 divided by the current internal level before the hourglass appears again.

The last thing to do is increase the variable **delay**. If you remember, **delay** is used to update the player movement and animation. If our program has finished, we kill the window and return to the desktop.

```
delay++;
}
}
// Shutdown
KillGLWindow();
return (msg.wParam);
```

// Increa

I spent a long time writing this tutorial. It started out as a simple line tutorial, and flourished into an entertaining mini game. Hopefully you can use what you have learned in this tutorial in GL projects of your own. I know alot of you have been asking about TILE based games. Well you can't get more tiled than this :) I've also gotten alot of emails asking how to do exact pixel plotting. I think I've got it covered :) Most importantly, this tutorial not only teaches you new things about OpenGL, it also teaches you how to use simple sounds to add excitement to your visual works of art! I hope you've enjoyed this tutorial. If you feel I have incorrectly commented something or that the code could be done better in some sections, please let me know. I want to make the best OpenGL tutorials I can and I'm interested in hearing your feedback.

Please note, this was an extremely large projects. I tried to comment everything as clearly as possible, but putting what things into words isn't as easy as it may seem. I know how everything works off by heart, but trying to explain is a different story :) If you've read through the tutorial and have a better way to word things, or if you feel diagrams might help out, please send me suggestions. I want this tutorial to be easy to follow through. Also note that this is not a beginner tutorial. If you haven't read through the previous tutorials please don't email me with questions until you have. Thanks.

Jeff Molofee (NeHe)

}

* DOWNLOAD Visual C++ Code For This Lesson.

Back To NeHe Productions!

Lesson 22

This lesson was written by Jens Schneider. It is loosely based on Lesson 06, though lots of changes were made. In this lesson you will learn:

- How to control your graphic-accelerator's multitexture-features.
- How to do a "fake" Emboss Bump Mapping.
- How to do professional looking logos that "float" above your rendered scene using blending.
- Basics about multi-pass rendering techniques.
- How to do matrix-transformations efficiently.

Since at least three of the above four points can be considered **"advanced rendering techniques"**, you should already have a general understanding of OpenGL's rendering pipeline. You should know most commands already used in these tutorials, and you should be familiar with vector-maths. Every now and then you'll encounter a block that reads **begin theory(...)** as header and **end theory(...)** as an ending. These sections try to teach you theory about the issue(s) mentioned in parenthesis. This is to ensure that, if you already know about the issue, you can easily skip them. If you encounter problems while trying to understand the code, consider going back to the theory sections. Last but not least: This lesson consists out of more than 1,200 lines of code, of which large parts are not only boring but also known among those that read earlier tutorials. Thus I will not comment each line, only the crux. If you encounter something like this >... <, it means that lines of code have been omitted.

Here we go:

#include <windows.h></windows.h>	
<pre>#include <stdio.h></stdio.h></pre>	// Header
<pre>#include <gl\gl.h></gl\gl.h></pre>	// Header
<pre>#include <gl\glu.h></gl\glu.h></pre>	
<pre>#include <gl\glaux.h></gl\glaux.h></pre>	
#include "glext.h"	// Header
<pre>#include <string.h></string.h></pre>	
<pre>#include <math.h></math.h></pre>	// Header

The **GLfloat MAX_EMBOSS** specifies the "strength" of the Bump Mapping-Effect. Larger values strongly enhance the effect, but reduce visual quality to the same extent by leaving so-called "artefacts" at the edges of the surfaces.

#define MAX_EMBOSS (GLfloat)0.01f

// Maximu

Ok, now let's prepare the use of the GL_ARB_multitexture extension. It's quite simple:

Most accelerators have more than just one texture-unit nowadays. To benefit of this feature, you'll have to check for **GL_ARB_multitexture**-support, which enables you to map two or more different textures to one OpenGL-primitive in just one pass. Sounds not too powerful, but it is! Nearly all the time if you're programming something, putting another texture on that object results in higher visual quality. Since you usually need multiple "**passes**" consisting out of interleaved texture-selection and drawing geometry, this can quickly become expensive. But don't worry, this will become clearer later on.

Now back to code: **___ARB_ENABLE** is used to override multitexturing for a special compile-run entirely. If you want to see your OpenGL-extensions, just un-comment the **#define EXT_INFO**. Next, we want to check for our extensions during run-time to ensure our code stays portable. So we need space for some strings. These are the following two lines. Now we want to distinguish between being able to do multitexture and using it, so we need another two flags. Last, we need to know how many texture-units are present(we're going to use only two of them, though). At least one texture-unit is present on any OpenGL-capable accelerator, so we initialize **maxTexelUnits** with 1.

#defineARB_ENABLE true	// Used T
// #define EXT_INFO	
#define MAX_EXTENSION_SPACE 10240	// Charac
#define MAX_EXTENSION_LENGTH 256	// Maximu
<pre>bool multitextureSupported=false;</pre>	// Flag I:
<pre>bool useMultitexture=true;</pre>	// Use It
GLint maxTexelUnits=1;	

The following lines are needed to "link" the extensions to C++ function calls. Just treat the **PFN**who-ever-reads-this as pre-defined datatype able to describe function calls. Since we are unsure if we'll get the functions to these prototypes, we set them to **NULL**. The commands **glMultiTexCoordifARB** map to the well-known **glTexCoordif**, specifying i-dimensional texturecoordinates. Note that these can totally substitute the **glTexCoordif**-commands. Since we only use the **GLfloat**-version, we only need prototypes for the commands ending with an "f". Other are also available ("fv", "i", etc.). The last two prototypes are to set the active texture-unit that is currently receiving texture-bindings (**glActiveTextureARB()**) and to determine which texture-unit is associated with the **ArrayPointer**-command (a.k.a. **Client-Subset**, thus **glClientActiveTextureARB**). By the way: **ARB** is an abbreviation for "**Architectural Review Board**". Extensions with **ARB** in their name are not required by an OpenGL-conformant implementation, but they are expected to be widely supported. Currently, only the multitextureextension has made it to **ARB**-status. This may be treated as sign for the tremendous impact regarding speed multitexturing has on several advanced rendering techniques.

The lines ommitted are GDI-context handles etc.

PFNGLMULTITEXCOORD1FARBPROC glMultiTexCoord1fARB= NULL;PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB= NULL;PFNGLMULTITEXCOORD3FARBPROC glMultiTexCoord3fARB= NULL;PFNGLMULTITEXCOORD4FARBPROC glMultiTexCoord4fARB= NULL;PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = NULL;

We need global variables:

- filter specifies what filter to use. Refer to Lesson 06. We'll usually just take GL_LINEAR, so we initialise with 1.
- texture holds our base-texture, three times, one per filter.
- **bump** holds our bump maps
- invbump holds our inverted bump maps. This is explained later on in a theory-section.
- The **Logo**-things hold textures for several billboards that will be added to rendering output as a final pass.
- The Light...-stuff contains data on our OpenGL light-source.

	// Which :
	// Our Bui
	// Handle
= { 0.2f, 0.2f, 0.2f};	
= { 1.0f, 1.0f, 1.0f};	
= { 0.0f, 0.0f, 2.0f};	
= { 0.5f, 0.5f, 0.5f, 1.0f};	
	= { 1.0f, 1.0f, 1.0f}; = { 0.0f, 0.0f, 2.0f};

The next block of code contains the numerical representation of a textured cube built out of **GL_QUADS**. Each five numbers specified represent one set of 2D-texture-coordinates one set of 3D-vertex-coordinates. This is to build the cube using for-loops, since we need that cube several times. The data-block is followed by the well-known **WndProc()**-prototype from former lessons.

// Data Contains The Faces Of The Cube In Format 2xTexCoord, 3xVertex.
// Note That The Tesselation Of The Cube Is Only Absolute Minimum.

```
GLfloat data[]= {
```

// FRONT FACE	
0.0f, 0.0f,	-1.0f, -1.0f, +1.0f,
1.0f, 0.0f,	+1.0f, -1.0f, +1.0f,
1.0f, 1.0f,	+1.0f, +1.0f, +1.0f,
0.0f, 1.0f,	-1.0f, +1.0f, +1.0f,
// BACK FACE	
1.0f, 0.0f,	-1.0f, -1.0f, -1.0f,
1.0f, 1.0f,	-1.0f, +1.0f, -1.0f,
0.0f, 1.0f,	+1.0f, +1.0f, -1.0f,
0.0f, 0.0f,	+1.0f, -1.0f, -1.0f,
// Top Face	
0.0f, 1.0f,	-1.0f, +1.0f, -1.0f,
0.0f, 0.0f,	-1.0f, +1.0f, +1.0f,
1.0f, 0.0f,	+1.0f, +1.0f, +1.0f,
1.0f, 1.0f,	+1.0f, +1.0f, -1.0f,
// Bottom Face	
1.0f, 1.0f,	-1.0f, -1.0f, -1.0f,
0.0f, 1.0f,	+1.0f, -1.0f, -1.0f,
0.0f, 0.0f,	+1.0f, -1.0f, +1.0f,
1.0f, 0.0f,	-1.0f, -1.0f, +1.0f,
// Right Face	
1.0f, 0.0f,	+1.0f, -1.0f, -1.0f,
1.0f, 1.0f,	+1.0f, +1.0f, -1.0f,

	0.0f, 1.0f,	+1.0f,	+1.0f,	+1.0f,
	0.0f, 0.0f,	+1.0f,	-1.0f,	+1.0f,
	// Left Face			
	0.0f, 0.0f,	-1.0f,	-1.0f,	-1.0f,
	1.0f, 0.0f,	-1.0f,	-1.0f,	+1.0f,
	1.0f, 1.0f,	-1.0f,	+1.0f,	+1.0f,
	0.0f, 1.0f,	-1.0f,	+1.0f,	-1.0f
;				

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

}

// Declara

The next block of code is to determine extension-support during run-time.

First, we can assume that we have a long string containing all supported extensions as '\n'seperated sub-strings. So all we need to do is to search for a '\n' and start comparing string with search until we encounter another '\n' or until string doesn't match search anymore. In the first case, return a true for "found", in the other case, take the next sub-string until you encounter the end of string. You'll have to watch a little bit at the beginning of string, since it does not begin with a newline-character.

By the way: A common rule is to ALWAYS check during runtime for availability of a given extension!

```
bool isInString(char *string, const char *search) {
         int pos=0;
         int maxpos=strlen(search)-1;
         int len=strlen(string);
         char *other;
         for (int i=0; i<len; i++) {</pre>
                  if ((i==0) || ((i>1) && string[i-1]=='\n')) {
                                                                                    // New Ex
                            other=&string[i];
                            pos=0;
                            while (string[i]!='\n') {
                                                                                    // Search
                                     if (string[i]==search[pos]) pos++;
                                                                                    // Next P
                                     if ((pos>maxpos) && string[i+1]=='\n') return true;
                                     i++;
                            }
                   }
         }
         return false;
}
```

Now we have to fetch the extension-string and convert it to be '\n'-separated in order to search it for our desired extension. If we find a sub-string "GL_ARB_multitexture" in it, this feature is supported. But we only can use it, if __ARB_ENABLE is also true. Last but not least we need GL_EXT_texture_env_combine to be supported. This extension introduces new ways how the texture-units interact. We need this, since GL_ARB_multitexture only feeds the output from one texture unit to the one with the next higher number. So we rather check for this extension than using another complex blending equation (that would not exactly do the same effect!) If all extensions are supported and we are not overridden, we'll first determine how much texture-units are available, saving them in maxTexelUnits. Then we have to link the functions to our names. This is done by the wglGetProcAdress()-calls with a string naming the function call as parameter and a prototype-cast to ensure we'll get the correct function type.

```
int len=strlen(extensions);
         for (int i=0; i<len; i++)</pre>
                                                                                    // Separa
                  if (extensions[i]==' ') extensions[i]='\n';
#ifdef EXT INFO
         MessageBox(hWnd,extensions,"supported GL extensions",MB_OK | MB_ICONINFORMATION);
#endif
         if (isInString(extensions, "GL_ARB_multitexture")
                                                                                    // Is Mul
                  && __ARB_ENABLE
                  && isInString(extensions, "GL_EXT_texture_env_combine"))
         {
                  glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB,&maxTexelUnits);
                  glMultiTexCoord1fARB = (PFNGLMULTITEXCOORD1FARBPROC) wglGetProcAddress("g
                  glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC) wglGetProcAddress("g
                  glMultiTexCoord3fARB = (PFNGLMULTITEXCOORD3FARBPROC) wglGetProcAddress("g
                  glMultiTexCoord4fARB = (PFNGLMULTITEXCOORD4FARBPROC) wglGetProcAddress("g
                  glactiveTextureARB = (PFNGLACTIVETEXTUREARBPROC) wglGetProcAddress("gla
                  glClientActiveTextureARB= (PFNGLCLIENTACTIVETEXTUREARBPROC) wglGetProcAdd:
#ifdef EXT_INFO
                  MessageBox(hWnd, "The GL_ARB_multitexture extension will be used.", "feature
#endif
                  return true;
         }
         useMultitexture=false;
         return false;
}
```

InitLights() just initialises OpenGL-Lighting and is called by InitGL() later on.

```
void initLights(void) {
    glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
    glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);
    glEnable(GL_LIGHT1);
}
```

Here we load LOTS of textures. Since auxDIBImageLoad() has an error-handler of it's own and since LoadBMP() wasn't much predictable without a try-catch-block, I just kicked it. But now to our loading-routine. First, we load the base-bitmap and build three filtered textures out of it (GL_NEAREST, GL_LINEAR and GL_LINEAR_MIPMAP_NEAREST). Note that I only use one data-structure to hold bitmaps, since we only need one at a time to be open. Over that I introduced a new data-structure called alpha here. It is to hold the alpha-layer of textures, so that I can save RGBA Images as two bitmaps: one 24bpp RGB and one 8bpp greyscale Alpha. For the status-indicator to work properly, we have to delete the Image-block after every load to reset it to NULL.

Note also, that I use **GL_RGB8** instead of just "3" when specifying texture-type. This is to be more conformant to upcoming OpenGL-ICD releases and should always be used instead of just another number. I marked it in orange for you.

```
int LoadGLTextures() {
    bool status=true;
    AUX_RGBImageRec *Image=NULL;
    char *alpha=NULL;
```

// Status

```
// Load The Tile-Bitmap for Base-Texture
if (Image=auxDIBImageLoad("Data/Base.bmp")) {
         glGenTextures(3, texture);
                                                                          // Create
         // Create Nearest Filtered Texture
         glBindTexture(GL_TEXTURE_2D, texture[0]);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, Image->sizeX, Image->sizeY, 0, GL
         // Create Linear Filtered Texture
         glBindTexture(GL_TEXTURE_2D, texture[1]);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, Image->sizeX, Image->sizeY, 0, GL
         // Create MipMapped Texture
         glBindTexture(GL_TEXTURE_2D, texture[2]);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAR
         gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB8, Image->sizeX, Image->sizeY, GL_
}
else status=false;
if (Image) {
        if (Image->data) delete Image->data;
                                                                          // If Tex
        delete Image;
         Image=NULL;
}
```

Now we'll load the Bump Map. For reasons discussed later, it has to have only 50% luminance, so we have to scale it in the one or other way. I chose to scale it using the **glPixelTransferf()**-commands, that specifies how data from bitmaps is converted to textures on pixel-basis. I use it to scale the RGB components of bitmaps to 50%. You should really have a look at the **glPixelTransfer()**-command family if you're not already using them in your programs. They're all quite useful.

Another issue is, that we don't want to have our bitmap repeated over and over in the texture. We just want it once, mapping to texture-coordinates (s,t)=(0.0f, 0.0f) thru (s,t)=(1.0f, 1.0f). All other texture-coordinates should be mapped to plain black. This is accomplished by the two glTexParameteri()-calls that are fairly self-explanatory and "clamp" the bitmap in s and t-direction.

```
// Load The Bumpmaps
if (Image=auxDIBImageLoad("Data/Bump.bmp")) {
         glPixelTransferf(GL_RED_SCALE,0.5f);
                                                                          // Scale 1
         glPixelTransferf(GL_GREEN_SCALE,0.5f);
         glPixelTransferf(GL_BLUE_SCALE,0.5f);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
                                                                          // No Wra
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
         glGenTextures(3, bump);
         // Create Nearest Filtered Texture
         >...<
         // Create Linear Filtered Texture
         >...<
         // Create MipMapped Texture
         >...<
```

You'll already know this sentence by now: For reasons discussed later, we have to build an inverted Bump Map, luminance at most 50% once again. So we subtract the bumpmap from pure white, which is **{255, 255, 255}** in integer representation. Since we do **NOT** set the RGB-Scaling back to **100%** (took me about three hours to figure out that this was a major error in my first version!), the inverted bumpmap will be scaled once again to 50% luminance.

```
for (int i=0; i<3*Image->sizeX*Image->sizeY; i++)
                                                                         // Invert
                  Image->data[i]=255-Image->data[i];
                                                                           // Create
         glGenTextures(3, invbump);
         // Create Nearest Filtered Texture
         >...<
         // Create Linear Filtered Texture
         >...<
         // Create MipMapped Texture
         >...<
}
else status=false;
if (Image) {
         if (Image->data) delete Image->data;
                                                                           // If Tex
         delete Image;
         Image=NULL;
}
```

Loading the Logo-Bitmaps is pretty much straightforward except for the RGB-A recombining, which should be self-explanatory enough for you to understand. Note that the texture is built from the **alpha**-memoryblock, not from the **Image**-memoryblock! Only one filter is used here.

```
// Load The Logo-Bitmaps
if (Image=auxDIBImageLoad("Data/OpenGL_ALPHA.bmp")) {
         alpha=new char[4*Image->sizeX*Image->sizeY];
         // Create Memory For RGBA8-Texture
         for (int a=0; a<Image->sizeX*Image->sizeY; a++)
                 alpha[4*a+3]=Image->data[a*3];
         if (!(Image=auxDIBImageLoad("Data/OpenGL.bmp"))) status=false;
         for (a=0; a<Image->sizeX*Image->sizeY; a++) {
                  alpha[4*a]=Image->data[a*3];
                                                                          // G
                  alpha[4*a+1]=Image->data[a*3+1];
                  alpha[4*a+2]=Image->data[a*3+2];
                                                                          // B
         }
         glGenTextures(1, &glLogo);
                                                                          // Create
         // Create Linear Filtered RGBA8-Texture
         glBindTexture(GL_TEXTURE_2D, glLogo);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
         glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Image->sizeX, Image->sizeY, 0, G
         delete alpha;
}
else status=false;
if (Image) {
```

}

```
// If Tex
         if (Image->data) delete Image->data;
         delete Image;
         Image=NULL;
}
// Load The "Extension Enabled"-Logo
if (Image=auxDIBImageLoad("Data/multi_on_alpha.bmp")) {
         alpha=new char[4*Image->sizeX*Image->sizeY];
                                                                            // Create
         >...<
         glGenTextures(1, &multiLogo);
         // Create Linear Filtered RGBA8-Texture
         > <
         delete alpha;
}
else status=false;
if (Image) {
                                                                            // If Tex
         if (Image->data) delete Image->data;
         delete Image;
         Image=NULL;
}
return status;
```

Next comes nearly the only unmodified function **ReSizeGLScene()**. I've omitted it here. It is followed by a function **doCube()** that draws a cube, complete with normalized normals. Note that this version only feeds texture-unit #0, since **glTexCoord2f(s,t)** is the same thing as **glMultiTexCoord2f(GL_TEXTURE0_ARB,s,t)**. Note also that the cube could be done using interleaved arrays, but this is definitely another issue. Note also that this cube **CAN NOT** be done using a display list, since display-lists seem to use an internal floating point accuracy different from **GLfloat**. Since this leads to several nasty effects, generally referred to as **"decaling"**-problems, I kicked display lists. I assume that a general rule for multipass algorithms is to do the entire geometry with or without display lists. So never dare mixing even if it seems to run on your hardware, since it won't run on any hardware!

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
// Resize And Initialize The GL Window
>....<
void doCube (void) {
         int i;
         glBegin(GL_QUADS);
                  // Front Face
                  glNormal3f( 0.0f, 0.0f, +1.0f);
                  for (i=0; i<4; i++) {
                            glTexCoord2f(data[5*i],data[5*i+1]);
                            glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
                   }
                   // Back Face
                  glNormal3f( 0.0f, 0.0f, -1.0f);
                  for (i=4; i<8; i++) {
                            glTexCoord2f(data[5*i],data[5*i+1]);
                            glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
                   // Top Face
                  glNormal3f( 0.0f, 1.0f, 0.0f);
                  for (i=8; i<12; i++) {
                            glTexCoord2f(data[5*i],data[5*i+1]);
                            glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
                   // Bottom Face
```

```
glNormal3f( 0.0f,-1.0f, 0.0f);
         for (i=12; i<16; i++) {</pre>
                   glTexCoord2f(data[5*i],data[5*i+1]);
                   glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
         }
         // Right Face
         glNormal3f( 1.0f, 0.0f, 0.0f);
         for (i=16; i<20; i++) {</pre>
                   glTexCoord2f(data[5*i],data[5*i+1]);
                   glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
         }
         // Left Face
         glNormal3f(-1.0f, 0.0f, 0.0f);
         for (i=20; i<24; i++) {
                   glTexCoord2f(data[5*i],data[5*i+1]);
                   glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
         }
glEnd();
```

```
}
```

Time to initialize OpenGL. All as in Lesson 06, except that I call **initLights()** instead of setting them here. Oh, and of course I'm calling Multitexture-setup, here!

int {	InitGL(GLvoid)	// All Se
	<pre>multitextureSupported=initMultitexture();</pre>	
	if (!LoadGLTextures()) return false;	// Jump T
	glEnable(GL_TEXTURE_2D);	// Enable
	glShadeModel(GL_SMOOTH);	// Enable
	glClearColor(0.0f, 0.0f, 0.0f, 0.5f);	
	glClearDepth(1.0f);	
	glEnable(GL_DEPTH_TEST);	// Enable
	glDepthFunc(GL_LEQUAL);	
	glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);	// Really
	<pre>initLights();</pre>	
	return true	

```
}
```

Here comes about 95% of the work. All references like "for reasons discussed later" will be solved in the following block of theory.

Begin Theory (Emboss Bump Mapping)

If you have a Powerpoint-viewer installed, it is highly recommended that you download the following presentation:

"Emboss Bump Mapping" by Michael I. Gold, nVidia Corp. [.ppt, 309K]

For those without Powerpoint-viewer, I've tried to convert the information contained in the document to .html-format. Here it comes:

Emboss Bump Mapping

Michael I. Gold

NVidia Corporation

Bump Mapping

Real Bump Mapping Uses Per-Pixel Lighting.

- Lighting calculation at each pixel based on perturbed normal vectors.
- Computationally expensive.
- For more information see: **Blinn, J. : Simulation of Wrinkled Surfaces**, Computer Graphics. 12,3 (August 1978) 286-292.
- For information on the web go to: <u>http://www.objectecture.com/</u> to see Cass Everitt's Orthogonal Illumination Thesis. (rem.: Jens)

Emboss Bump Mapping

Emboss Bump Mapping Is A Hack

- Diffuse lighting only, no specular component
- Under-sampling artefacts (may result in blurry motion, rem.: Jens)
- Possible on today's consumer hardware (as shown, rem.: Jens)
- If it looks good, do it!

Diffuse Lighting Calculation

C=(L*N) x Dl x Dm

- L is light vector
- N is normal vector
- **Dl** is light diffuse color
- **Dm** is material diffuse color
- Bump Mapping changes N per pixel
- Emboss Bump Mapping approximates (L*N)

Approximate Diffuse Factor L*N

Texture Map Represents Heightfield

- [0,1] represents range of bump function
- First derivate represents slope **m** (Note that **m** is only 1D. Imagine **m** to be the inf.-norm of **grad(s,t)** to a given set of coordinates (**s,t**)!, **rem.: Jens**)
- m increases / decreases base diffuse factor Fd
- (Fd+m) approximates (L*N) per pixel

<u>Approximate Derivative</u>

Embossing Approximates Derivative

- Lookup height **H0** at point (**s**,**t**)
- Lookup height **H1** at point slightly perturbed toward light source (s+ds,t+dt)
- Subtract original height H0 from perturbed height H1
- Difference represents instantaneous slope **m=H1-H0**

Compute The Bump



1) Original bump (H0).



2) Original bump (H0) overlaid with second bump (H1) slightly perturbed toward light source.



3) Substract original bump from second (H0-H1). This leads to brightened (B) and darkened (D) areas.

Compute The Lighting

Evaluate Fragment Color Cf

- $Cf = (L*N) \times Dl \times Dm$
- $(L^*N) \sim (Fd + (H1-H0))$
- **Dm x Dl** is encoded in surface texture **Ct**. Could control **Dl** seperately, if you're clever. (we control it using OpenGL-Lighting!, **rem.: Jens**)
- $Cf = (Fd + (H0-H1) \times Ct)$

Is That All? It's So Easy!

We're Not Quite Done Yet. We Still Must:

- Build a texture (using a painting program, **rem.: Jens**)
- Calculate texture coordinate offsets (ds,dt)
- Calculate diffuse Factor **Fd** (is controlled using OpenGL-Lighting!, **rem.: Jens**)
- Both are derived from normal N and light vector L (in our case, only (ds,dt) are calculated explicitly!, rem.: Jens)
- Now we have to do some math

Building A Texture

Conserve Textures!

- Current multitexture-hardware only supports two textures! (By now, not true anymore, but nevertheless you should read this!, **rem.: Jens**)
- Bump Map in **ALPHA** channel (not the way we do it, could implement it yourself as an exercise if you have TNT-chipset **rem.: Jens**)
- Maximum bump = 1.0
- Level ground = 0.5
- Maximum depression = 0.0
- Surface color in **RGB** channels
- Set internal format to **GL_RGBA8** !!

Calculate Texture Offsets

Rotate Light Vector Into Normal Space

- Need Normal coordinate system
- Derive coordinate system from normal and "up" vector (we pass the texCoord directions to our offset generator explicitly, **rem.: Jens**)
- Normal is z-axis
- Cross-product is x-axis
- Throw away "up" vector, derive y-axis as cross-product of x- and z-axis
- Build 3x3 matrix Mn from axes
- Transform light vector into normal space.(Mn is also called an orthonormal basis. Think of Mn*v as

to "express" v in means of a basis describing tangent space rather than in means of the standard basis. Note also that orthonormal bases are invariant against-scaling resulting in no loss of normalization when multiplying vectors! **rem.: Jens**)

Calculate Texture Offsets (Cont'd)

Use Normal-Space Light Vector For Offsets

- $L' = Mn \times L$
- Use L'x, L'y for (ds,dt)
- Use L'z for diffuse factor! (Rather not! If you're no TNT-owner, use OpenGL-Lighting instead, since you have to do one additional pass anyhow!, rem.: Jens)
- If light vector is near normal, **L'x**, **L'y** are small.
- If light vector is near tangent plane, L'x, L'y are large.
- What if **L'z** is less than zero?
- Light is on opposite side from normal
- Fade contribution toward zero.

Implementation On TNT

Calculate Vectors, Texcoords On The Host

- Pass diffuse factor as vertex alpha
- Could use vertex **color** for light diffuse color
- **HO** and surface color from texture unit 0
- **H1** from texture unit 1 (same texture, different coordinates)
- **ARB_multitexture** extension
- **Combines** extension (more precisely: the **NVIDIA_multitexture_combiners extension**, featured by all TNT-family cards, **rem.: Jens**)

Implementation on TNT (Cont'd)

Combiner 0 Alpha-Setup:

- (1-T0a) + T1a 0.5 (T0a stands for "texture-unit 0, alpha channel", rem.: Jens)
- (T1a-T0a) maps to (-1,1), but hardware clamps to (0,1)
- 0.5 bias balances the loss from clamping (consider using 0.5 scale, since you can use a wider variety of bump maps, **rem.: Jens**)
- Could modulate light diffuse color with **T0c**
- Combiner 0 rgb-setup:
- (T0c * C0a + T0c * Fda 0.5)*2
- 0.5 bias balances the loss from clamping
- scale by 2 brightens the image

End Theory (Emboss Bump Mapping)

Though we're doing it a little bit different than the TNT-Implementation to enable our program to run on **ALL** accelerators, we can learn two or three things here. One thing is, that bump mapping is a multi-pass algorithm on most cards (not on TNT-family, where it can be implemented in one 2-texture pass.) You should now be able to imagine how nice multitexturing really is. We'll now implement a 3-pass non-multitexture algorithm, that can be (and will be) developed into a 2-pass multitexture algorithm.

By now you should be aware, that we'll have to do some matrix-matrix-multiplication (and matrix-vector-multiplication, too). But that's nothing to worry about: OpenGL will do the matrix-matrix-multiplication for us (if tweaked right) and the matrix-vector-multiplication is really easy-going: VMatMult(M,v) multiplies matrix M with vector v and stores the result back in v: v:=M*v. All Matrices and vectors passed have to be in homogenous-coordinates resulting in 4x4 matrices and 4-dim vectors. This is to ensure conformity to OpenGL in order to multiply own vectors with OpenGL-matrices right away.

```
// Calculates v=vM, M Is 4x4 In Column-Major, v Is 4dim. Row (i.e. "Transposed")
void VMatMult(GLfloat *M, GLfloat *v) {
    GLfloat res[3];
    res[0]=M[ 0]*v[0]+M[ 1]*v[1]+M[ 2]*v[2]+M[ 3]*v[3];
    res[1]=M[ 4]*v[0]+M[ 5]*v[1]+M[ 6]*v[2]+M[ 7]*v[3];
    res[2]=M[ 8]*v[0]+M[ 9]*v[1]+M[10]*v[2]+M[11]*v[3];
    v[0]=res[0];
    v[1]=res[1];
    v[2]=res[2];
    v[3]=M[15];
}
```

Begin Theory (Emboss Bump Mapping Algorithms)

Here we'll discuss two different algorithms. I found the first one several days ago under: http://www.nvidia.com/marketing/Developer/DevRel.nsf/TechnicalDemosFrame?OpenPage

The program is called **GL_BUMP** and was written by Diego Tártara in 1999. It implements really nice looking bump mapping, though it has some drawbacks. But first, lets have a look at Tártara's Algorithm:

- 1. All vectors have to be EITHER in object OR world space
- 2. Calculate vector v from current vertex to light position
- 3. Normalize v
- 4. Project v into tangent space. (This is the plane touching the surface in the current vertex. Typically, if working with flat surfaces, this is the surface itself).
- 5. Offset (s,t)-coordinates by the projected v's x and y component

This looks not bad! It is basically the Algorithm introduced by **Michael I. Gold** above. But it has a major drawback: Tártara only does the projection for a **xy**-plane! This is not sufficient for our purposes since it simplifies the projection step to just taking the xy-components of v and discarding the z-component.

But his implementation does the diffuse lighting the same way we'll do it: by using OpenGL's builtin lighting. Since we can't use the combiners-method Gold suggests (we want our programs to run anywhere, not just on TNT-cards!), we can't store the diffuse factor in the alpha channel. Since we already have a 3-pass non-multitexture / 2-pass multitexture problem, why not apply OpenGL-Lighting to the last pass to do all the ambient light and color stuff for us? This is possible (and looks quite well) only because we have no complex geometry, so keep this in mind. If you'd render several thousands of bump mapped triangles, try to invent something new!

Furthermore, he uses multitexturing, which is, as we shall see, not as easy as you might have thought regarding this special case.

But now to our Implementation. It looks quite the same to the above Algorithm, except for the projection step, where we use an own approach:

• We use **OBJECT COORDINATES**, this means we don't apply the modelview matrix to our calculations. This has a nasty side-effect: since we want to rotate the cube, object-coordinates of the cube don't change, world-coordinates (also referred to as eye-coordinates) do. But our light-position should not be rotated with the cube, it should be just static, meaning that it's world-coordinates don't change. To compensate, we'll apply a trick commonly used in computer graphics: Instead of transforming each vertex to worldspace in advance to computing the bumps, we'll just transform the light into object-space by applying the inverse of the modelview-matrix. This is very cheap in this case since we know exactly how the modelview-matrix was built step-by-step, so an inversion can also be done step-by-

step. We'll come back later to that issue.

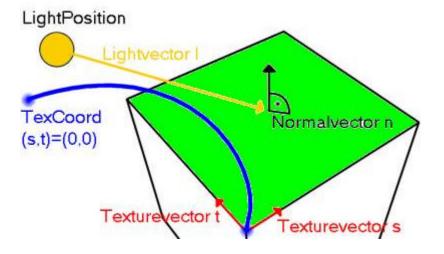
- We calculate the **current vertex c** on our surface (simply by looking it up in **data**).
- Then we'll calculate a normal **n** with length 1 (We usually know **n** for each face of a cube!). This is important, since we can save computing time by requesting normalized vectors. Calculate the **light vector v** from **c** to the **light position l**
- If there's work to do, build a matrix **Mn** representing the orthonormal projection. This is done as f
- Calculate out texture coordinate offset by multiplying the supplied texture-coordinate directions s and t each with v and MAX_EMBOSS: ds = s*v*MAX_EMBOSS, dt=t*v*MAX_EMBOSS. Note that s, t and v are vectors while MAX_EMBOSS isn't.
- Add the offset to the texture-coordinates in pass 2.

Why this is good:

- Fast (only needs one squareroot and a couple of MULs per vertex)!
- Looks very nice!
- This works with all surfaces, not just planes.
- This runs on all accelerators.
- Is glBegin/glEnd friendly: Does not need any "forbidden" GL-commands.

Drawback:

- Not fully physical correct.
- Leaves minor artefacts.



This figure shows where our vectors are located. You can get t and s by simply subtracting adjacent vertices, but be sure to have them point in the right direction and to normalize them. The blue spot marks the vertex where texCoord2f(0.0f,0.0f) is mapped to.

End Theory (Emboss Bump Mapping Algorithms)

Let's have a look to texture-coordinate offset generation, first. The function is called **SetUpBumps** (), since this actually is what it does:

// Sets Up The Texture-Offsets
// n : Normal On Surface. Must Be Of Length 1
// c : Current Vertex On Surface
// l : Lightposition
// s : Direction Of s-Texture-Coordinate In Object Space (Must Be Normalized!)
// t : Direction Of t-Texture-Coordinate In Object Space (Must Be Normalized!)
void SetUpBumps(GLfloat *n, GLfloat *c, GLfloat *l, GLfloat *s, GLfloat *t) {

```
GLfloat v[3];
GLfloat lenQ;
// Calculate v From Current Vertex c To Lightposition And Normalize v
v[0]=1[0]-c[0];
v[1]=1[1]-c[1];
v[2]=1[2]-c[2];
lenQ=(GLfloat) sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
v[0]/=lenQ;
v[1]/=lenQ;
v[1]/=lenQ;
v[2]/=lenQ;
// Project v Such That We Get Two Values Along Each Texture-Coordinate Axis
c[0]=(s[0]*v[0]+s[1]*v[1]+s[2]*v[2])*MAX_EMBOSS;
c[1]=(t[0]*v[0]+t[1]*v[1]+t[2]*v[2])*MAX_EMBOSS;
```

Doesn't look that complicated anymore, eh? But theory is necessary to understand and control this effect. (I learned **THAT** myself during writing this tutorial).

I always like logos to be displayed while presentational programs are running. We'll have two of them right now. Since a call to **doLogo()** resets the **GL_MODELVIEW**-matrix, this has to be called as final rendering pass.

This function displays two logos: An OpenGL-Logo and a multitexture-Logo, if this feature is enabled. The logos are alpha-blended and are sort of semi-transparent. Since they have an alpha-channel, I blend them using **GL_SRC_ALPHA**, **GL_ONE_MINUS_SRC_ALPHA**, as suggested by all OpenGL-documentation. Since they are all co-planar, we do not have to z-sort them before. The numbers that are used for the vertices are "empirical" (a.k.a. try-and-error) to place them neatly into the screen edges. We'll have to enable blending and disable lighting to avoid nasty effects. To ensure they're in front of all, just reset the **GL_MODELVIEW**-matrix and set depth-function to **GL_ALWAYS**.

```
void doLogo(void) {
         // MUST CALL THIS LAST !!!, Billboards The Two Logos
        glDepthFunc(GL_ALWAYS);
        glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
         glEnable(GL_BLEND);
         glDisable(GL_LIGHTING);
         glLoadIdentity();
         glBindTexture(GL_TEXTURE_2D,glLogo);
         glBegin(GL_QUADS);
                  glTexCoord2f(0.0f,0.0f); glVertex3f(0.23f, -0.4f,-1.0f);
                  glTexCoord2f(1.0f,0.0f); glVertex3f(0.53f, -0.4f,-1.0f);
                  glTexCoord2f(1.0f,1.0f); glVertex3f(0.53f, -0.25f,-1.0f);
                  glTexCoord2f(0.0f,1.0f); glVertex3f(0.23f, -0.25f,-1.0f);
         glEnd();
         if (useMultitexture) {
                  glBindTexture(GL_TEXTURE_2D,multiLogo);
                  glBegin(GL_QUADS);
                           glTexCoord2f(0.0f,0.0f); glVertex3f(-0.53f, -0.25f,-1.0f);
                           glTexCoord2f(1.0f,0.0f); glVertex3f(-0.33f, -0.25f,-1.0f);
                           glTexCoord2f(1.0f,1.0f); glVertex3f(-0.33f, -0.15f,-1.0f);
                           glTexCoord2f(0.0f,1.0f); glVertex3f(-0.53f, -0.15f,-1.0f);
                  glEnd();
         }
}
```

Here comes the function for doing the bump mapping without multitexturing. It's a three-pass implementation. As a first step, the **GL_MODELVIEW** matrix is inverted by applying to the identity-matrix all steps later applied to the **GL_MODELVIEW** in reverse order and inverted. The result is a matrix that "undoes" the **GL_MODELVIEW** if applied to an object. We fetch it from OpenGL by simply using **glGetFloatv()**. Remember that the matrix has to be an array of 16 and that the matrix is "transposed"!

By the way: If you don't exactly know how the modelview was built, consider using world-space, since matrix-inversion is complicated and costly. But if you're doing large amounts of vertices inverting the modelview with a more generalized approach could be faster.

```
bool doMesh1TexelUnits(void) {
         GLfloat c[4]={0.0f,0.0f,0.0f,1.0f};
                                                                                    // Holds (
         GLfloat n[4]={0.0f,0.0f,0.0f,1.0f};
                                                                                    // Normal
         GLfloat s[4]={0.0f,0.0f,0.0f,1.0f};
                                                                                    // s-Text
         GLfloat t[4]={0.0f,0.0f,0.0f,1.0f};
                                                                                    // t-Text
         GLfloat 1[4];
         GLfloat Minv[16];
                                                                                    // Holds '
         int i;
         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                                                                    // Clear '
         // Build Inverse Modelview Matrix First. This Substitutes One Push/Pop With One gll
         // Simply Build It By Doing All Transformations Negated And In Reverse Order
         glLoadIdentity();
         glRotatef(-yrot,0.0f,1.0f,0.0f);
         glRotatef(-xrot,1.0f,0.0f,0.0f);
         glTranslatef(0.0f, 0.0f, -z);
         glGetFloatv(GL_MODELVIEW_MATRIX,Minv);
         glLoadIdentity();
         glTranslatef(0.0f,0.0f,z);
         glRotatef(xrot,1.0f,0.0f,0.0f);
         glRotatef(yrot, 0.0f, 1.0f, 0.0f);
         // Transform The Lightposition Into Object Coordinates:
         l[0]=LightPosition[0];
         l[1]=LightPosition[1];
         l[2]=LightPosition[2];
         1[3]=1.0f;
         VMatMult(Minv,l);
```

First Pass:

- Use bump-texture
- Disable Blending
- Disable Lighting
- Use non-offset texture-coordinates
- Do the geometry

This will render a cube only consisting out of bump map.

```
glBindTexture(GL_TEXTURE_2D, bump[filter]);
glDisable(GL_BLEND);
glDisable(GL_LIGHTING);
doCube();
```

Second Pass:

- Use inverted bump-texture
- Enable Blending GL_ONE, GL_ONE
- Keep Lighting disabled
- Use offset texture-coordinates (This means that you call SetUpBumps() before each face of the cube
- Do the geometry

This will render a cube with the correct emboss bump mapping, but without colors.

You could save computing time by just rotating the lightvector into inverted direction. However, this didn't work out correctly, so we do it the plain way: rotate each normal and center-point the same way we rotate our geometry!

```
glBindTexture(GL_TEXTURE_2D, invbump[filter]);
glBlendFunc(GL_ONE,GL_ONE);
glDepthFunc(GL_LEQUAL);
glEnable(GL_BLEND);
glBegin(GL_QUADS);
         // Front Face
         n[0]=0.0f;
         n[1]=0.0f;
         n[2]=1.0f;
         s[0]=1.0f;
         s[1]=0.0f;
         s[2]=0.0f;
         t[0]=0.0f;
         t[1]=1.0f;
         t[2]=0.0f;
         for (i=0; i<4; i++) {
                  c[0]=data[5*i+2];
                  c[1]=data[5*i+3];
                  c[2]=data[5*i+4];
                  SetUpBumps(n,c,l,s,t);
                  glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
                  glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
         }
         // Back Face
         n[0]=0.0f;
         n[1]=0.0f;
         n[2]=-1.0f;
         s[0]=-1.0f;
         s[1]=0.0f;
         s[2]=0.0f;
         t[0]=0.0f;
         t[1]=1.0f;
         t[2]=0.0f;
         for (i=4; i<8; i++) {
                  c[0]=data[5*i+2];
                  c[1]=data[5*i+3];
                  c[2]=data[5*i+4];
                  SetUpBumps(n,c,l,s,t);
                  glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
                  glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
         }
         // Top Face
         n[0]=0.0f;
```

```
n[1]=1.0f;
n[2]=0.0f;
s[0]=1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=0.0f;
t[2]=-1.0f;
for (i=8; i<12; i++) {
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
         c[2]=data[5*i+4];
         SetUpBumps(n,c,l,s,t);
         glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
         glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// Bottom Face
n[0]=0.0f;
n[1] = -1.0f;
n[2]=0.0f;
s[0]=-1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=0.0f;
t[2]=-1.0f;
for (i=12; i<16; i++) {
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
         c[2]=data[5*i+4];
         SetUpBumps(n,c,l,s,t);
         glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
         glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// Right Face
n[0]=1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
s[2]=-1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=16; i<20; i++) {</pre>
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
         c[2]=data[5*i+4];
         SetUpBumps(n,c,l,s,t);
         glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
         glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
// Left Face
n[0] = -1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
s[2]=1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=20; i<24; i++) {
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
```

```
c[2]=data[5*i+4];
SetUpBumps(n,c,l,s,t);
glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
glEnd();
```

Third Pass:

- Use (colored) base-texture
- Enable Blending GL_DST_COLOR, GL_SRC_COLOR
- This blending equation multiplies by 2: (Cdst*Csrc)+(Csrc*Cdst)=2(Csrc*Cdst)!
- Enable Lighting to do the ambient and diffuse stuff
- Reset GL_TEXTURE-matrix to go back to "normal" texture coordinates
- Do the geometry

This will finish cube-rendering, complete with lighting. Since we can switch back and forth between multitexturing and non-multitexturing, we have to reset the texture-environment to "normal" **GL_MODULATE** first. We only do the third pass, if the user doesn't want to see just the emboss.

```
if (!emboss) {
    glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glBindTexture(GL_TEXTURE_2D,texture[filter]);
    glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
    glEnable(GL_LIGHTING);
    doCube();
}
```

Last Pass:

- update geometry (esp. rotations)
- do the Logos

```
xrot+=xspeed;
yrot+=yspeed;
if (xrot>360.0f) xrot-=360.0f;
if (xrot<0.0f) xrot+=360.0f;
if (yrot>360.0f) yrot-=360.0f;
if (yrot<0.0f) yrot+=360.0f;
/* LAST PASS: Do The Logos! */
doLogo();
return true;
```

}

This function will do the whole mess in 2 passes with multitexturing support. We support two texelunits. More would be extreme complicated due to the blending equations. Better trim to TNT instead. Note that almost the only difference to **doMesh1TexelUnits()** is, that we send two sets of texture-coordinates for each vertex!

```
GLfloat c[4]={0.0f,0.0f,0.0f,1.0f};
                                                                          // Holds (
GLfloat n[4]={0.0f,0.0f,0.0f,1.0f};
                                                                          // Normal
GLfloat s[4]={0.0f,0.0f,0.0f,1.0f};
                                                                          // s-Text
GLfloat t[4]={0.0f,0.0f,0.0f,1.0f};
                                                                          // t-Text
GLfloat 1[4];
GLfloat Minv[16];
                                                                          // Holds '
int i;
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                                                           // Clear '
// Build Inverse Modelview Matrix First. This Substitutes One Push/Pop With One gll
// Simply Build It By Doing All Transformations Negated And In Reverse Order
glLoadIdentity();
glRotatef(-yrot,0.0f,1.0f,0.0f);
glRotatef(-xrot,1.0f,0.0f,0.0f);
glTranslatef(0.0f,0.0f,-z);
glGetFloatv(GL_MODELVIEW_MATRIX,Minv);
glLoadIdentity();
glTranslatef(0.0f,0.0f,z);
glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);
// Transform The Lightposition Into Object Coordinates:
l[0]=LightPosition[0];
l[1]=LightPosition[1];
l[2]=LightPosition[2];
1[3]=1.0f;
VMatMult(Minv,l);
```

First Pass:

- No Blending
- No Lighting

Set up the texture-combiner 0 to

- Use bump-texture
- Use not-offset texture-coordinates
- Texture-Operation GL_REPLACE, resulting in texture just being drawn

Set up the texture-combiner 1 to

- Offset texture-coordinates
- Texture-Operation GL_ADD, which is the multitexture-equivalent to ONE, ONE- blending.

This will render a cube consisting out of the grey-scale erode map.

```
// TEXTURE-UNIT #0
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, bump[filter]);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);
// TEXTURE-UNIT #1
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, invbump[filter]);
```

glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT); glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_ADD); // General Switches glDisable(GL_BLEND); glDisable(GL_LIGHTING);

Now just render the faces one by one, as already seen in **doMesh1TexelUnits()**. Only new thing: Uses **glMultiTexCoor2fARB()** instead of just **glTexCoord2f()**. Note that you must specify which texture-unit you mean by the first parameter, which must be **GL_TEXTUREi_ARB** with i in [0..31]. (What hardware has 32 texture-units? And what for?)

```
glBegin(GL_QUADS);
         // Front Face
         n[0]=0.0f;
         n[1]=0.0f;
         n[2]=1.0f;
         s[0]=1.0f;
         s[1]=0.0f;
         s[2]=0.0f;
         t[0]=0.0f;
         t[1]=1.0f;
         t[2]=0.0f;
         for (i=0; i<4; i++) {
                  c[0]=data[5*i+2];
                  c[1]=data[5*i+3];
                  c[2]=data[5*i+4];
                  SetUpBumps(n,c,l,s,t);
                  glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
                  glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]
                  glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
         }
         // Back Face
         n[0]=0.0f;
         n[1]=0.0f;
         n[2]=-1.0f;
         s[0]=-1.0f;
         s[1]=0.0f;
         s[2]=0.0f;
         t[0]=0.0f;
         t[1]=1.0f;
         t[2]=0.0f;
         for (i=4; i<8; i++) {
                  c[0]=data[5*i+2];
                  c[1]=data[5*i+3];
                  c[2]=data[5*i+4];
                  SetUpBumps(n,c,l,s,t);
                  glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
                  glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]
                  glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
         }
         // Top Face
         n[0]=0.0f;
         n[1]=1.0f;
         n[2]=0.0f;
         s[0]=1.0f;
         s[1]=0.0f;
         s[2]=0.0f;
         t[0]=0.0f;
         t[1]=0.0f;
         t[2]=-1.0f;
```

```
for (i=8; i<12; i++) {
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
         c[2]=data[5*i+4];
         SetUpBumps(n,c,l,s,t);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
         glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]
         glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// Bottom Face
n[0]=0.0f;
n[1]=-1.0f;
n[2]=0.0f;
s[0]=-1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=0.0f;
t[2]=-1.0f;
for (i=12; i<16; i++) {
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
         c[2]=data[5*i+4];
         SetUpBumps(n,c,l,s,t);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
         glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]
         glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
// Right Face
n[0]=1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
s[2]=-1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=16; i<20; i++) {</pre>
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
         c[2]=data[5*i+4];
         SetUpBumps(n,c,l,s,t);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
         glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]
         glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
// Left Face
n[0]=-1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
s[2]=1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=20; i<24; i++) {
         c[0]=data[5*i+2];
         c[1]=data[5*i+3];
         c[2]=data[5*i+4];
         SetUpBumps(n,c,l,s,t);
         glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
         glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]
         glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
```

```
glEnd();
```

Second Pass

• Use the base-texture

}

- Enable Lighting
- No offset texturre-coordinates => reset GL_TEXTURE-matrix
- Reset texture environment to GL_MODULATE in order to do OpenGLLighting (doesn't work otherwise!)

This will render our complete bump-mapped cube.

```
glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);
if (!emboss) {
    glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glBindTexture(GL_TEXTURE_2D,texture[filter]);
    glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
    glEnable(GL_BLEND);
    glEnable(GL_LIGHTING);
    doCube();
}
```

Last Pass

}

- Update Geometry (esp. rotations)
- Do The Logos

```
xrot+=xspeed;
yrot+=yspeed;
if (xrot>360.0f) xrot-=360.0f;
if (xrot<0.0f) xrot+=360.0f;
if (yrot>360.0f) yrot-=360.0f;
if (yrot<0.0f) yrot+=360.0f;
/* LAST PASS: Do The Logos! */
doLogo();
return true;
```

Finally, a function to render the cube without bump mapping, so that you can see what difference this makes!

```
bool doMeshNoBumps(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear '
    glLoadIdentity(); // Reset '
    glTranslatef(0.0f,0.0f,z);
    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);
```

```
if (useMultitexture) {
         glActiveTextureARB(GL_TEXTURE1_ARB);
         glDisable(GL_TEXTURE_2D);
         glActiveTextureARB(GL_TEXTURE0_ARB);
}
glDisable(GL_BLEND);
glBindTexture(GL_TEXTURE_2D,texture[filter]);
glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
glEnable(GL_LIGHTING);
doCube();
xrot+=xspeed;
yrot+=yspeed;
if (xrot>360.0f) xrot-=360.0f;
if (xrot<0.0f) xrot+=360.0f;
if (yrot>360.0f) yrot-=360.0f;
if (yrot<0.0f) yrot+=360.0f;
/* LAST PASS: Do The Logos! */
doLogo();
return true;
```

All the drawGLScene() function has to do is to determine which doMesh-function to call:

Kills the GLWindow, not modified (thus omitted):

GLvoid KillGLWindow(GLvoid)
>...<</pre>

}

Creates the GLWindow, not modified (thus omitted):

BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag) >...<

Windows main-loop, not modified (thus omitted):

// Here's

LRESULT CALLBACK WndProc(HWND hWnd, UINT

UINT uMsg, WPARAM wParam, LPARAM lParam)

>...<

Windows main-function, added some keys:

- E: Toggle Emboss / Bumpmapped Mode
- M: Toggle Multitexturing
- B: Toggle Bumpmapping. This Is Mutually Exclusive With Emboss Mode
- F: Toggle Filters. You'll See Directly That GL_NEAREST Isn't For Bumpmapping
- CURSOR-KEYS: Rotate The Cube

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, // Previo LPSTR lpCmdLine, // Comman int nCmdShow) { >...< if (keys['E']) { keys['E']=false; emboss=!emboss; } if (keys['M']) { keys['M']=false; useMultitexture=((!useMultitexture) && multite } if (keys['B']) { keys['B']=false; bumps=!bumps; } if (keys['F']) { keys['F']=false; filter++; filter%=3; } if (keys[VK_PRIOR]) { z-=0.02f; } if (keys[VK_NEXT]) { z+=0.02f; } if (keys[VK_UP])

// Handle

```
{
                                       xspeed-=0.01f;
                             }
                             if (keys[VK_DOWN])
                              {
                                       xspeed+=0.01f;
                             }
                             if (keys[VK_RIGHT])
                              {
                                       yspeed+=0.01f;
                             }
                             if (keys[VK_LEFT])
                              {
                                       yspeed-=0.01f;
                             }
                    }
          }
// Shutdown
KillGLWindow();
return (msg.wParam);
```

Now that you managed this tutorial some words about generating textures and bumpmapped objects before you start to program mighty games and wonder why bumpomapping isn't that fast or doesn't look that good:

- You shouldn't use textures of 256x256 as done in this lesson. This slows things down a lot. Only do so if demonstrating visual capabilities (like in tutorials).
- A bumpmapped cube is not usual. A rotated cube far less. The reason for this is the viewing angle: The steeper it gets, the more visual distortion due to filtering you get. Nearly all multipass algorithms are very affected by this. To avoid the need for high-resolution textures, reduce the minimum viewing angle to a sensible value or reduce the bandwidth of viewing angles and pre-filter you texture to perfectly fit that bandwidth.
- You should first have the colored-texture. The bumpmap can be often derived from it using an average paint-program and converting it to grey-scale.
- The bumpmap should be "sharper" and higher in contrast than the color-texture. This is usually done by applying a "sharpening filter" to the texture and might look strange at first, but believe me: you can sharpen it **A LOT** in order to get first class visual appearance.
- The bumpmap should be centered around 50%-grey (RGB=127,127,127), since this means "no bump at all", brighter values represent ing bumps and lower "scratches". This can be achieved using "histogram" functions in some paint-programs.
- The bumpmap can be one fourth in size of the color-texture without "killing" visual appearance, though you'll definitely see the difference.

Now you should at least have a basic understanding of the issued covered in this tutorial. I hope you have enjoyed reading it.

If you have questions and / or suggestions regarding this lesson, you can just **mail me**, since I have not yet a web page.

This is my current project and will follow soon.

Thanks must go to:

}

- Michael I. Gold for his Bump Mapping Documentation
- **Diego Tártara** for his example code
- NVidia for putting great examples on the WWW

• And last but not least to NeHe who helped me learn a lot about OpenGL.

Jens Schneider Jeff Molofee (NeHe)

* DOWNLOAD Visual C++ Code For This Lesson.

Back To NeHe Productions!

Lesson 23

Advanced Input with Direct Input and Windows

With the way things are nowadays, you must use the latest technology to compete with games such as Quake and Unreal. In this tutorial, I will teach you how to set up your compiler for Direct Input, how to use it, and how to use the mouse in Opengl w/ Windows. This tutorial is based on code from Lesson 10. So open the Lesson 10 source code and lets get started!

The Mouse

First we need to add in a variable to hold the mouse's X and Y position.

```
typedef struct tagSECTOR
{
    int numtriangles;
    TRIANGLE* triangle;
} SECTOR;
SECTOR sector1;
POINT mpos;
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
// Declaration For
```

Ok, as you can see, we have added in a new variable called **mpos**. (Mouse Position). **mpos** has two variables, **x** and **y**. We will use these variables to figure out how to rotate the scene. We will modify parts of CreateGLWindow() with the following code.

ShowCursor(FALSE);	// Hide Mouse Point	
if (fullscreen)	// Are We	
{		
dwExStyle=WS_EX_APPWINDOW;	// Window Extended	
dwStyle=WS_POPUP;	// Windows Style	
}		

Above, we moved the ShowCursor() out of the if statement below it. Therefore, if we go fullscreen or windowed the cursor will never be shown. Now we need to get and set the mouse every time we render. So modify the following in WinMain():

```
SwapBuffers(hDC);
GetCursorPos(&mpos);
SetCursorPos(320,240);
heading += (float)(320 - mpos.x)/100 * 5;
yrot = heading;
lookupdown -= (float)(240 - mpos.y)/100 * 5;
```

Lots to talk about here. First we get the mouse position with GetCursorPos(POINT p). This tells us how much to rotate on the X and Y axis. After we've got the position, we set it up for the next rendering pass using SetCursorPos(int X, int Y).

Note: Do not set the mouse position to 0,0! If you do, you will not be able to move the mouse to the upper left because 0,0 is the upper left of the window. 320 is the middle of the window from left to right and 240 is the middle of the window from the top to the bottom in 640x480 mode. Just a reminder!

After we have taken care of the mouse we need to update some stuff for rendering and movement.

float = (P - CX) / U * S;

- **P** The point we set the mouse to every time
- CX The current mouse position
- U Units
- **S** Mouse speed (being a hardcore quaker I like it at 12)

We also do this for the **heading** variable and the **lookupdown** variable.

There you have it! Mouse code worthy of the greats!

The KeyBoard (DirectX 7)

Now we can look around in our world. The next step is to read multiple keys. By adding this section of code, you will be able to walk forward, strafe, and crouch all at the same time! Enough chit-chat, let's code!

I will now explain the steps required to use DirectX 7. The first step will depend on your compiler. I will show you how to do it with Visual C++, although it shouldn't be much different with other compilers.

- First you must download, or order, the DirectX 7 Sdk (128 MB). You can download or order by clicking here Or click here to download the necessary library and include files locally (1.32 MB). Make sure you have DX7 installed on your computer.
- Next you must install the sdk or necessary on your computer. If you are using the dx7.zip file from this site, all you have to do is unzip the file, and move all of the include files into your Visual C++ include directory, and all of the library files into your Visual C++ library directory. The Visual C++ directories can usually be found at C:\Program Files\Microsoft Visual Studio\VC98. If you are not using Visual Studio, look for a directory called Visual C. Hopefully by now you know where the library files are, and where the include files are.
- 3. After you have installed the required files, open your project and go to **Project->Settings**.
- 4. Click on the Link tab, and move down to Object / Library Modules.
- 5. Type in the following at the beginning of the line, **dinput.lib dxguid.lib winmm.lib**. This links the Direct Input library, the DirectX GUI library, and the Windows Multimedia library (required for timing code) into our program.

Now we have Direct Input setup for compiling in our project. Time for some coding!

We need to include the Direct Input header file so that we can use some of its functions. Also, we need to add in Direct Input and the Direct Input Keyboard Device.

#include <stdio.h></stdio.h>		// Header File For
<pre>#include <gl\gl.h></gl\gl.h></pre>		// Header File For
#include <gl\glu.h></gl\glu.h>		// Header
<pre>#include <gl\glaux.h></gl\glaux.h></pre>		// Header
<pre>#include <dinput.h></dinput.h></pre>		// Direct
LPDIRECTINPUT7	g_DI;	// Direct
LPDIRECTINPUTDEVICE7	g_KDIDev;	// Keyboard Device

The last two lines above set up Direct Input (g_DI) and the the keyboard device (g_KDIDev). The keyboard device receives the input and we translate and use it. Direct Input is not too far off from regular windows input as seen in the previous tutorials.

Windows	Direct Input
VK_LEFT	DIK_LEFT
VK_RIGHT	DIK_RIGHT
etc	

Basically all we do is change VK to DIK. Although I think that some keys have changed.

Now we need to add a new function to setup Direct Input and the keyboard device. Underneath CreateGLWindow(), add the following:

```
// Initializes Direct Input ( Add )
int DI_Init()
{
         // Create Direct Input
         // Window Instance
                                            DIRECTINPUT_VERSION,// DIID_IDirectInput7,// Version 7(void**)&g_DI,// DNULL )// NULL Parametee
                                                                         // Direct
                                                                                 // Direct
                                                                     // NULL Parameter
                                             NULL ) )
         {
                 return(false);
                                                                                  // Couldn
         }
         // Create The Keyboard Device
         if ( q_DI->CreateDeviceEx( GUID_SysKeyboard, // Define Which Device Tto (
                                             IID_IDirectInputDevice7, // Version 7
                                             (void**)&g_KDIDev, // KeyBoard Device
NULL ) ) // NULL Parameter
         {
                 return(false);
                                                                                  // Couldn
         }
         // Set The Keyboard Data Format
         if ( q_KDIDev->SetDataFormat(&c_dfDIKeyboard) )
         {
                                                                                  // Could 1
                 return(false);
         }
         // Set The Cooperative Level
         if ( g_KDIDev->SetCooperativeLevel(hWnd, DISCL_FOREGROUND | DISCL_EXCLUSIVE) )
         {
```

Jeff Molofee's OpenGL Windows Tutorial #23 (By Justin Eslinger)

```
return(false);
                                                                                       // Could 1
         }
                                                                                       // Did We
         if (g_KDIDev)
                                                                                       // If So,
                   g_KDIDev->Acquire();
                                                                                       // If Not
         else
                   return(false);
                                                                                       // Return
         return(true);
                                                                                       // Everyt]
}
// Destroys DX ( Add )
void DX_End()
{
         if (g_DI)
         {
                   if (g_KDIDev)
                   {
                            g_KDIDev->Unacquire();
                             g_KDIDev->Release();
                            g_KDIDev = NULL;
                   }
                   g_DI->Release();
                   g_DI = NULL;
         }
}
```

I think the code above is pretty self explanatory. First we init Direct Input, then we create the Keyboard device, and finally we acquire it. Later on, I might talk about how you can also use the Mouse and Joystick with Direct Input (although I don't suggest using Direct Input for the mouse).

Now we need to change the code from Window's Input to Direct Input. Which means a whole lot of modifying! So, here we go!

In WndProc() The Following Code Was Removed

```
// Is A Key Being I
case WM_KEYDOWN:
{
         keys[wParam] = TRUE;
                                                                   // If So,
         return 0;
                                                          // Jump Back
}
case WM_KEYUP:
                                                                   // Has A 1
{
         kevs[wParam] = FALSE;
                                                                   // If So,
         return 0;
                                                          // Jump Back
}
```

At The Top Of The Program, Make The Following Changes

BYTE	buffer[256];	// New Ke
bool	active=TRUE;	// Window
bool	fullscreen=TRUE;	// Fullscreen Flag
bool	blend;	// Blendi
bool	bp;	// Blend 1
bool	fp;	// Fl Key

• • •

GLfloat lookupdown = 0.0f;

Jeff Molofee's OpenGL Windows Tutorial #23 (By Justin Eslinger)

```
GLfloat z=0.0f;
                                                                                 // Depth
GLuint filter;
                                                                                 // Which 1
GLuint texture[5];
                                                                        // Storage For 5 Te
In The WinMain() Function
         // Create Our OpenGL Window
         if (!CreateGLWindow("Justin Eslinger's & NeHe's Advanced DirectInput Tutorial",640
         {
                 return 0;
                                                                        // Quit If Window V
         }
         if (!DI_Init())
                                                                                 // Initia
         {
                  return 0;
         }
         . . .
                           // Draw The Scene. Watch For ESC Key And Quit Messages From Draw
                           if ((active && !DrawGLScene()))
                                                                          // Active
                           . . .
                                    HRESULT hr = g_KDIDev->GetDeviceState(sizeof(buffer), &
                                    if ( buffer[DIK_ESCAPE] & 0x80 ) // Check For Escape
                                    {
                                             done=TRUE;
                                    }
                                    if ( buffer[DIK_B] & 0x80) // B Key Being Pre:
                                    {
                                             if (!bp)
                                             {
                                                                               // Is The
                                                      bp = true;
                                                      blend=!blend;
                                                      if (!blend)
                                                      {
                                                               glDisable(GL_BLEND);
                                                               glEnable(GL_DEPTH_TEST);
                                                      }
                                                      else
                                                      {
                                                               glEnable(GL_BLEND);
                                                               glDisable(GL_DEPTH_TEST);
                                                      }
                                             }
                                    }
                                    else
                                    {
                                             bp = false;
                                    }
                                    if ( buffer[DIK_PRIOR] & 0x80 ) // Page U
                                    {
                                             z-=0.02f;
                                    }
                                    if ( buffer[DIK_NEXT] & 0x80 )
                                                                                 // Page D
                                    {
                                             z+=0.02f;
                                    }
```

```
if ( buffer[DIK_UP] & 0x80 )
                                                // Up Arr
{
         xpos -= (float)sin(heading*piover180) * 0.05f;
         zpos -= (float)cos(heading*piover180) * 0.05f;
         if (walkbiasangle >= 359.0f)
          {
                   walkbiasangle = 0.0f;
         }
         else
         {
                   walkbiasangle+= 10;
          }
         walkbias = (float)sin(walkbiasangle * piover18
}
if ( buffer[DIK_DOWN] & 0x80 )
                                                // Down A:
         xpos += (float)sin(heading*piover180) * 0.05f;
         zpos += (float)cos(heading*piover180) * 0.05f;
         if (walkbiasangle <= 1.0f)</pre>
          {
                   walkbiasangle = 359.0f;
         }
         else
         {
                   walkbiasangle-= 10;
          }
         walkbias = (float)sin(walkbiasangle * piover18
}
if ( buffer[DIK_LEFT] & 0x80 )
                                                // Left A:
{
         xpos += (float)sin((heading - 90)*piover180) *
         zpos += (float)cos((heading - 90)*piover180) *
         if (walkbiasangle <= 1.0f)
          {
                   walkbiasangle = 359.0f;
         }
         else
         {
                   walkbiasangle-= 10;
          }
         walkbias = (float)sin(walkbiasangle * piover18
}
if ( buffer[DIK_RIGHT] & 0x80 )
                                                // Right 1
{
         xpos += (float)sin((heading + 90)*piover180) *
         zpos += (float)cos((heading + 90)*piover180) *
         if (walkbiasangle <= 1.0f)</pre>
          {
                   walkbiasangle = 359.0f;
         }
         else
         {
                   walkbiasangle-= 10;
          }
         walkbias = (float)sin(walkbiasangle * piover18
}
if ( buffer[DIK_F1] & 0x80)
                                      // Is F1 Being Pres
```

```
{
                                                if (!fp)
                                                                            // If F1 Isn't Bei
                                                {
                                                         fp = true;
                                                                                      // Is The
                                                         KillGLWindow();
                                                                                      // Kill O
                                                         fullscreen=!fullscreen;
                                                                                      // Toggle
                                                         // Recreate Our OpenGL Window
                                                         if (!CreateGLWindow("Justin Eslinger'
                                                         {
                                                                   return 0;// Quit If Window V
                                                         }
                                                         if (!DI_Init())
                                                                                      // ReInit
                                                         {
                                                                   return 0;// Couldn't Initial
                                                         }
                                                }
                                      }
                                      else
                                      {
                                                fp = false;
                                                                                      // Set 'f]
                                      }
                            }
                   }
         }
         // Shutdown
         DX_End();
                                                                            // Destroys Directl
                                                                                     // Kill T
         KillGLWindow();
                                                                                      // Exit Tl
         return (msg.wParam);
}
In DrawGLScene() Modify From The First Line Below
         glTranslatef(xtrans, ytrans, ztrans);
         numtriangles = sector1.numtriangles;
         // Process Each Triangle
         for (int loop_m = 0; loop_m < numtriangles; loop_m++)</pre>
         {
```

glBindTexture(GL_TEXTURE_2D, texture[sector1.triangle[loop_m].texture]);

```
glBegin(GL_TRIANGLES);
```

Ok, I need to discuss some things here. First, I took out some stuff that we didn't need. Then I replaced the old Windows keyboard stuff. I also changed the left and right keys. Now, when you go left or right, it strafes instead of turns! All I did was add or minus 90 degrees from the heading direction! That's pretty much it! Everything else is commented. Now we can compile and run our game! Whoohooo! hehehe

Now we have Direct Input and Mouse support in our game, what next? Well, we need to add in a timing system to regulate the speed in our game. Without timing, we check the input every frame and that could make us zip through the level before we can even look at it! So let's get started!

First we need adjustment variables to slow down our game and a timer structure.

POINT mpos; int adjust = 5;

// Create A Structure For The Timer Information (Add)

// Mouse : // Speed :

```
struct
{
           __int64
                            frequency;
                                                                                             // Timer :
          float
                              resolution;
                                                                                              // Timer 1
         unsigned long mm_timer_start;
unsigned long mm_timer_elapsed;
bool performance_timer
                                                                                              // Multime
                                                                                   // Multimedia Time:
                                                                                   // Using The Perfor
                              performance_timer;
          __int64
                              performance_timer_start;
                                                                                   // Performance Time
          int64
                              performance_timer_elapsed;
                                                                                   // Performance Time
} timer;
                                                                                   // Structure Is Nam
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
                                                                                   // Declaration For
```

The above code was discussed in lesson 21 so I shouldn't have to explain it.

Just below the declaration for the wndproc(), we need to add in the timer functions.

```
// Initialize Our Timer (Get It Ready) ( Add )
void TimerInit(void)
{
         memset(&timer, 0, sizeof(timer));
                                                                          // Clear Our Timer
         // Check To See If A Performance Counter Is Available
         // If One Is Available The Timer Frequency Will Be Updated
         if (!QueryPerformanceFrequency((LARGE_INTEGER *) &timer.frequency))
         {
                  // No Performace Counter Available
                  timer.performance_timer = FALSE;
                                                                          // Set Performance
                  timer.mm_timer_start = timeGetTime();
                                                                                   // Use ti
                  timer.resolution = 1.0f/1000.0f;
                                                                          // Set Our Timer Re
                  timer.frequency = 1000;
                                                                                   // Set Ou:
                  timer.mm_timer_elapsed = timer.mm_timer_start;
                                                                                   // Set The
         }
         else
         {
                  // Performance Counter Is Available, Use It Instead Of The Multimedia Tim
                  // Get The Current Time And Store It In performance_timer_start
                  QueryPerformanceCounter((LARGE_INTEGER *) &timer.performance_timer_start)
                  timer.performance_timer = TRUE;
                                                                                   // Set Pe:
                  // Calculate The Timer Resolution Using The Timer Frequency
                  timer.resolution = (float) (((double)1.0f)/((double)timer.frequency));
                  // Set The Elapsed Time To The Current Time
                  timer.performance_timer_elapsed = timer.performance_timer_start;
         }
}
// Get Time In Milliseconds ( Add )
float TimerGetTime()
{
          int64 time;
                                                                                   // time W
         if (timer.performance_timer)
                                                                                   // Are We
         {
                  QueryPerformanceCounter((LARGE_INTEGER *) &time);
                                                                          // Grab The Current
                  // Return The Current Time Minus The Start Time Multiplied By The Resolut
                  return ( (float) ( time - timer.performance_timer_start) * timer.resoluti
         }
         else
         {
                  // Return The Current Time Minus The Start Time Multiplied By The Resolut.
                  return( (float) ( timeGetTime() - timer.mm_timer_start) * timer.resolutio:
```

}

}

The above was also in Lesson 21 so nothing to explain here. Just make sure you add the winmm.lib library file. Otherwise you will get errors when you compile.

Now we must add some stuff in the WinMain() function.

```
if (!DI_Init())
                                                                            // Initia
{
         return 0;
}
TimerInit();
                                                                            // Init O
. . .
                   float start=TimerGetTime();
                   // Grab Timer Value Before We Draw ( Add )
                   // Draw The Scene. Watch For ESC Key And Quit Messages From Dra
                   if ((active && !DrawGLScene()))
                                                                           // Active
                   {
                            done=TRUE;
                                                                           // ESC or
                   }
                                                                            // Not Tii
                   else
                   {
                            while(TimerGetTime()<start+float(adjust*2.0f)) {}</pre>
```

Now the game will run at the adjusted speed. The following segment will be dedicated to some graphical adjustments I made to the Lesson 10 Level.

If you've already downloaded the code for this lesson, then you've already seen that I've added multiple textures to the scene. The new textures are in the DATA directory. Here's how I added the textures:

Modify the tagTriangle structure

```
Modify the SetupWorld code
```

```
for (int loop = 0; loop < numtriangles; loop++)
{
     readstr(filein,oneline); ( Add )
     sscanf(oneline, "%i\n", &sector1.triangle[loop].texture); ( Add )
     for (int vert = 0; vert < 3; vert++)
     {
</pre>
```

Modify the DrawGLScene code

```
// Process Each Triangle
for (int loop_m = 0; loop_m < numtriangles; loop_m++)</pre>
```

```
{
   glBindTexture(GL_TEXTURE_2D, texture[sector1.triangle[loop_m].texture]);
   glBegin(GL_TRIANGLES);
```

```
In the LoadGLTextures Code We Add More Textures
```

```
int LoadGLTextures()
                                                                                    // Load B
                                                                          // Status Indicato
         int Status=FALSE;
                                                                          // Create Storage :
         AUX_RGBImageRec *TextureImage[5];
         memset(TextureImage,0,sizeof(void *)*2);
                                                                          // Set The Pointer
                  (TextureImage[0]=LoadBMP("Data/floor1.bmp")) &&
                                                                                   // Load T
         if (
                                                                                   // Load T
                  (TextureImage[1]=LoadBMP("Data/light1.bmp")) &&
                  (TextureImage[2]=LoadBMP("Data/rustyblue.bmp")) &&
                                                                          // Load the Wall Te
                  (TextureImage[3]=LoadBMP("Data/crate.bmp")) &&
                                                                                   // Load Tl
                  (TextureImage[4]=LoadBMP("Data/weirdbrick.bmp")))
                                                                          // Load the Ceiling
                  Status=TRUE;
                                                                                    // Set Th
                  glGenTextures(5, &texture[0]);
                                                                                    // Create
                  for (int loop1=0; loop1<5; loop1++)</pre>
                                                                          // Loop Through 5 1
                  {
                           glBindTexture(GL_TEXTURE_2D, texture[loop1]);
                           glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop1]->sizeX, Te
                                     GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop1]->data);
                           glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
                           glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                  for (loop1=0; loop1<5; loop1++)</pre>
                                                                                    // Loop T
                  {
                           if (TextureImage[loop1])
                                                                          // If Texture Exist
                            {
                                     if (TextureImage[loop1]->data)
                                                                                    // If Tex
                                     {
                                              free(TextureImage[loop1]->data);// Free The Te
                                     free(TextureImage[loop1]);
                                                                 // Free The Image :
                            }
         return Status;
                                                                                    // Return
}
```

So now you're able to harness the awesome power of Direct Input. I spent a lot of time writing and revising this tutorial to make it very easy to understand and also error free. I hope this tutorial is helpful to those that wanted to learn this. I felt that since this site gave me enough knowledge to create the engine I have now, that I should give back to the community. Thanks for taking the time to read this!

Justin Eslinger (BlackScar)

blackscar@ticz.com http://members.xoom.com/Blackscar/

* DOWNLOAD Visual C++ Code For This Lesson.

Back To NeHe Productions!

Lesson 24

Sphere Environment Mapping is a quick way to add a reflection to a metallic or reflective object in your scene. Although it is not as accurate as real life or as a Cube Environment Map, it is a whole lot faster! We'll be using the code from lesson eighteen (Quadratics) for the base of this tutorial. Also we're not using any of the same texture maps, we're going to use one sphere map, and one background image.

Before we start... The "red book" defines a Sphere map as a picture of the scene on a metal ball from infinite distance away and infinite focal point. Well that is impossible to do in real life. The best way I have found to create a good sphere map image without using a Fish eye lens is to use Adobe's Photoshop program.

Creating a Sphere Map In Photoshop:

First you will need a picture of the environment you want to map onto the sphere. Open the picture in Adobe Photoshop and select the entire image. Copy the image and create a new PSD (Photoshop Format) the new image should be the same size as the image we just copied. Paste a copy of the image into the new window we've created. The reason we make a copy is so Photoshop can apply its filters. Instead of copying the image you can select mode from the drop down menu and choose RGB mode. All of the filters should then be available.

Next we need to resize the image so that the image dimensions are a power of 2. Remember that in order to use an image as a texture the image needs to be 128x128, 256x256, etc. Under the image menu, select image size, uncheck the constraint proportions checkbox, and resize the image to a valid texture size. If your image is 100X90, it's better to make the image 128x128 than 64x64. Making the image smaller will lose alot of detail.

The last thing we do is select the filter menu, select distort and apply a spherize modifier. You should see that the center of the picture is blown up like a balloon, now in normal sphere maps the outer area will be blackened out, but it doesn't really matter. Save a copy of the image as a .BMP and you're ready to code!

We don't add any new global variables this time but we do modify the texture array to hold 6 textures.

GLuint texture[6];

1,

The next thing I did was modify the LoadGLTextures() function so we can load in 2 bitmaps and create 3 filters. (Like we did in the original texturing tutorials). Basically we loop through twice and create 3 textures each time using a different filtering mode. Almost all of this code is new or modified.

int {	LoadGLTextures()	/ ,
C C	int Status=FALSE;	// Status :
	AUX_RGBImageRec *TextureImage[2];	// Create :
	<pre>memset(TextureImage,0,sizeof(void *)*2);</pre>	// Set The

Jeff Molofee's OpenGL Windows Tutorial #24 (By GB Schmick (TipTup))

```
// Load The Bitmap, Check For Errors, If Bitmap's Not Found Quit
if ((TextureImage[0]=LoadBMP("Data/BG.bmp")) &&
                                                                                   1,
         (TextureImage[1]=LoadBMP("Data/Reflect.bmp")))
                                                                                   1,
{
         Status=TRUE;
                                                                                   1,
         glGenTextures(6, &texture[0]);
         for (int loop=0; loop<=1; loop++)</pre>
         Ł
                  // Create Nearest Filtered Texture
                  glBindTexture(GL_TEXTURE_2D, texture[loop]);
                                                                          // Gen Tex
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
                  glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, Te>
                  // Create Linear Filtered Texture
                  glBindTexture(GL_TEXTURE_2D, texture[loop+2]);
                                                                                   1,
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
                  glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, Tex
                  // Create MipMapped Texture
                  glBindTexture(GL_TEXTURE_2D, texture[loop+4]);
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MI
                  gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[loop]->sizeX, 1
         for (loop=0; loop<=1; loop++)</pre>
        if (TextureImage[loop])
             ł
                          if (TextureImage[loop]->data)
                                                                                   1,
                                {
                                             free(TextureImage[loop]->data);
                                                                                   1,
                                     free(TextureImage[loop]);
                                                                         // Free The
                  }
         }
}
return Status;
                                                                                   1,
```

We'll modify the cube drawing code a little. Instead of using 1.0 and -1.0 for the normal values, we'll use 0.5 and -0.5. By changing the value of the normal, you can zoom the reflection map in and out. If the normal value is high, the image being reflected will be bigger, and may appear blocky. By reducing the normal value to 0.5 and -0.5 the reflected image is zoomed out a bit so that the image reflecting off the cube isn't all blocky looking. Setting the normal value too low will create undesirable results.

}

```
GLvoid glDrawCube()
                  glBegin(GL_QUADS);
                  // Front Face
                  glNormal3f( 0.0f, 0.0f, 0.5f);
                 glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,
                                                                     1.0f);
                 glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
                  glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
                  glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f,
                                                                     1.0f);
                  // Back Face
                  glNormal3f( 0.0f, 0.0f, -0.5f);
                                                                                          (
                 glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
                 glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
                 glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
                 glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
                  // Top Face
```

Jeff Molofee's OpenGL Windows Tutorial #24 (By GB Schmick (TipTup))

```
glNormal3f( 0.0f, 0.5f, 0.0f);
                                                                                                             (
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,
                                                                        1.0f, -1.0f);
            // Bottom Face
            glNormal3f( 0.0f,-0.5f, 0.0f);
                                                                                                             (
            glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
            // Right Face
            glNormal3f( 0.5f, 0.0f, 0.0f);
                                                                                                             (
            glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
            // Left Face
            glNormal3f(-0.5f, 0.0f, 0.0f);
                                                                                                             (
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
```

```
}
```

Now in InitGL we add two new function calls, these two calls set the texture generation mode for S and T to Sphere Mapping. The texture coordinates S, T, R & Q relate in a way to object coordinates x, y, z and w. If you are using a one-dimensional texture (1D) you will use the S coordinate. If your texture is two dimensional, you will use the S & T coordinates.

So what the following code does is tells OpenGL how to automatically generate the S and T coordinates for us based on the sphere-mapping formula. The R and Q coordinates are usually ignored. The Q coordinate can be used for advanced texture mapping extensions, and the R coordinate may become useful once 3D texture mapping has been added to OpenGL, but for now we will ignore the R & Q Coords. The S coordinate runs horizontally across the face of our polygon, the T coordinate runs vertically across the face of our polygon.

glTexGeni(GL_S,	GL_TEXTURE_GEN_MODE,	GL_SPHERE_MAP);	17	Set	The
glTexGeni(GL_T,	GL_TEXTURE_GEN_MODE,	GL_SPHERE_MAP);	11	Set	The

We're almost done! All we have to do is set up the rendering, I took out a few of the quadratic objects because they didn't work well with environment mapping. The first thing we need to do is enable texture generation. Then we select the reflective texture (sphere map) and draw our object. After all of the objects you want sphere-mapped have been drawn, you will want to disable texture generation, otherwise everything will be sphere mapped. We disable sphere-mapping before we draw the background scene (we don't want the background sphere mapped). You will notice that the bind texture commands may look fairly complex. All we're doing is selecting the filter to use when drawing our sphere map or the background image.

```
int DrawGLScene(GLvoid) /,
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear TI
    glLoadIdentity(); // Reset TI
    glTranslatef(0.0f,0.0f,z);
    glEnable(GL_TEXTURE_GEN_S); //,
    glEnable(GL_TEXTURE_GEN_T); //,
    glBindTexture(GL_TEXTURE_2D, texture[filter+(filter+1)]); // This Wi:
```

Jeff Molofee's OpenGL Windows Tutorial #24 (By GB Schmick (TipTup))

```
glPushMatrix();
glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);
switch(object)
case 0:
          glDrawCube();
          break;
case 1:
          glTranslatef(0.0f,0.0f,-1.5f);
                                                                                             1.
          gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32);
                                                                                   // A Cylind
          break;
case 2:
          gluSphere(quadratic, 1.3f, 32, 32);
                                                                                   // Sphere I
          break;
case 3:
          glTranslatef(0.0f,0.0f,-1.5f);
          gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32);
                                                                                   // Cone Wit
          break;
};
glPopMatrix();
glDisable(GL_TEXTURE_GEN_S);
                                                                                             1,
glDisable(GL_TEXTURE_GEN_T);
                                                                                             1.
glBindTexture(GL_TEXTURE_2D, texture[filter*2]);
                                                                                   // This Wil
glPushMatrix();
          glTranslatef(0.0f, 0.0f, -24.0f);
          glBegin(GL_QUADS);
                    glNormal3f( 0.0f, 0.0f, 1.0f);
                    glTexCoord2f(0.0f, 0.0f); glVertex3f(-13.3f, -10.0f, 10.0f);
                    glTexCoord2f(1.0f, 0.0f); glVertex3f(13.3f, -10.0f,
glTexCoord2f(1.0f, 1.0f); glVertex3f(13.3f, 10.0f,
glTexCoord2f(0.0f, 1.0f); glVertex3f(-13.3f, 10.0f,
                                                                                   10.0f);
                                                                                   10.0f);
                                                                                   10.0f);
          glEnd();
glPopMatrix();
xrot+=xspeed;
yrot+=yspeed;
return TRUE;
                                                                                             1,
```

The last thing we have to do is update the spacebar section of code to reflect (No Pun Intended) the changes we made to the Quadratic objects being rendered. (We removed the discs)

```
if (keys[' '] && !sp)
{
            sp=TRUE;
            object++;
            if(object>3)
                 object=0;
}
```

We're done! Now you can do some really impressive things with Environment mapping like making an almost accurate reflection of a room! I was planning on showing how to do Cube Environment Mapping in this tutorial too but my current video card does not support cube mapping. Maybe in a month or so after I buy a GeForce 2 :) Also I taught myself environment mapping (mostly because I couldnt find too much information on it) so if anything in this tutorial is inaccurate, Email Me or let NeHe know.

Thanks, and Good Luck!

}

GB Schmick (**TipTup**)

Jeff Molofee's OpenGL Windows Tutorial #24 (By GB Schmick (TipTup))

tiptup@net4tv.com http://www.tiptup.com/

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)

Back To NeHe Productions!

Lesson 25

This tutorial is far from visually stunning, but you will definitely learn a few new things by reading through it. I have had quite a few people ask me about extensions, and how to find out what extensions are supported on a particular brand of video card. This tutorial will teach you how to find out what OpenGL extensions are supported on any type of 3D video card.

I will also teach you how to scroll a portion of the screen without affecting any of the graphics around it using scissor testing. You will also learn how to draw line strips, and most importantly, in this tutorial we will drop the AUX library completely, along with Bitmap images. I will show you how to use Targa (TGA) images as textures. Not only are Targa files easy to work with and create, they support the ALPHA channel, which will allow you to create some pretty cool effects in future projects!

The first thing you should notice in the code below is that we no longer include the glaux header file (glaux.h). It is also important to note that the glaux.lib file can also be left out! We're not working with bitmaps anymore, so there's no need to include either of these files in our project.

Also, using glaux, I always received one warning message. Without glaux there should be zero errors, zero warnings.

<pre>#include <windows.h> // Header File For W #include <stdio.h> // Header File For Sta #include <stdarg.h> // Header File For Va #include <string.h> // Header File For St #include <gl\gl.h> // Header File For The #include <gl\glu.h> // Header File For The</gl\glu.h></gl\gl.h></string.h></stdarg.h></stdio.h></windows.h></pre>	ndard Input / Output riable Argument Routines ring Management OpenGL32 Library
HDC hDC=NULL; // Private GDI Device Co	ntext
HGLRC hRC=NULL; // Permanent Rendering	Context
HWND hWnd=NULL; // Holds Our Window Ha	ndle
HINSTANCE hInstance; // Holds The Instance	e Of The Application
heel here [256]: // America Heed Here	Varbaard Daubing
bool keys[256]; // Array Used For The	
bool active=TRUE; // Window Active Fla	g Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Fl	ag Set To Fullscreen Mode By Default

The first thing we need to do is add some variables. The first variable **scroll** will be used to scroll a portion of the screen up and down. The second variable **maxtokens** will be used to keep track of how many tokens (extensions) are supported by the video card.

base is used to hold the font display list.

int	scroll; //	Used For Scrolling The Screen
int	maxtokens;	// Keeps Track Of The Number Of Extensions Supported
GLuint	base; /	/ Base Display List For The Font

Now we create a structure to hold the TGA information once we load it in. The first variable **imageData** will hold a pointer to the data that makes up the image. **bpp** will hold the bits per pixel

Jeff Molofee's OpenGL Windows Tutorial #25

used in the TGA file (this value should be 24 or 32 bits depending on whether or not there is an alpha channel). The third variable **width** will hold the width of the TGA image. **height** will hold the height of the image, and **texID** will be used to keep track of the textures once they are built. The structure will be called **TextureImage**.

The line just after the structure (**TextureImage textures[1]**) sets aside storage for the one texture that we will be using in this program.

```
typedef struct // Create A Structure
{
   GLubyte *imageData; // Image Data (Up To 32 Bits)
   GLuint bpp; // Image Color Depth In Bits Per Pixel
   GLuint width; // Image Width
   GLuint height; // Image Height
   GLuint texID; // Texture ID Used To Select A Texture
} TextureImage; // Structure Name
TextureImage textures[1]; // Storage For One Texture
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration For WndProc
```

Now for the fun stuff! This section of code will load in a TGA file and convert it into a texture for use in the program. One thing to note is that this code will only load 24 or 32 bit uncompressed TGA files. I had a hard enough time making the code work with both 24 and 32 bit TGA's :) I never said I was a genious. I'd like to point out that I did not write all of this code on my own. Alot of the really good ideas I got from reading through random sites on the net. I just took all the good ideas and combined them into code that works well with OpenGL. Not easy, not extremely difficult!

We pass two parameters to this section of code. The first parameter points to memory that we can store the texture in (***texture**). The second parameter is the name of the file that we want to load (***filename**).

The first variable **TGAheader[]** holds 12 bytes. We'll compare these bytes with the first 12 bytes we read from the TGA file to make sure that the file is indeed a Targa file, and not some other type of image.

TGAcompare will be used to hold the first 12 bytes we read in from the TGA file. The bytes in **TGAcompare** will then be compared with the bytes in **TGAheader** to make sure everything matches.

header[] will hold the first 6 IMPORTANT bytes from the header file (width, height, and bits per pixel).

The variable **bytesPerPixel** will store the result after we divide bits per pixel by 8, leaving us with the number of bytes used per pixel.

imageSize will store the number of bytes required to make up the image (width * height * bytes per pixel).

temp is a temporary variable that we will use to swap bytes later in the program.

The last variable **type** is a variable that I use to select the proper texture building params depending on whether or not the TGA is 24 or 32 bit. If the texture is 24 bit we need to use GL_RGB mode when we build the texture. If the TGA is 32 bit we need to add the Alpha component, meaning we have to use GL_RGBA (By default I assume the image is 32 bit by default that is why **type** is GL_RGBA).

bool LoadTGA(Te	xtureImage *texture, char *filename) // Loads A To	GA File Into Memory
{		
GLubyte	$TGAheader[12] = \{0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}; // Uncor$	npressed TGA Header
GLubyte	TGAcompare[12]; // Used To Compare TGA Header	

Jeff Molofee's OpenGL Windows Tutorial #25

GLubyteheader[6]; // First 6 Useful Bytes From The HeaderGLuintbytesPerPixel; // Holds Number Of Bytes Per Pixel Used In The TGA FileGLuintimageSize; // Used To Store The Image Size When Setting Aside RamGLuinttemp; // Temporary VariableGLuinttype=GL_RGBA; // Set The Default GL Mode To RBGA (32 BPP)

The first line below opens the TGA file for reading. **file** is the handle we will use to point to the data within the file. the command fopen(**filename**, "rb") will open the file **filename**, and "rb" tells our program to open it for [r]eading in [b]inary mode!

The if statement has a few jobs. First off it checks to see if the file contains any data. If there is no data, NULL will be returned, the file will be closed with fclose(file), and we return false.

If the file contains information, we attempt to read the first 12 bytes of the file into **TGAcompare**. We break the line down like this: fread will read sizeof(**TGAcompare**) (12 bytes) from **file** into **TGAcompare**. Then we check to see if the number of bytes read is equal to sizeof(**TGAcompare**) which should be 12 bytes. If we were unable to read the 12 bytes into **TGAcompare** the file will close and false will be returned.

If everything has gone good so far, we then compare the 12 bytes we read into **TGAcompare** with the 12 bytes we have stored in **TGAheader**. If the bytes do not match, the file will close, and false will be returned.

Lastly, if everything has gone great, we attempt to read 6 more bytes into **header** (the important bytes). If 6 bytes are not available, again, the file will close and the program will return false.

```
FILE *file = fopen(filename, "rb"); // Open The TGA File
if( file==NULL || // Does File Even Exist?
  fread(TGAcompare,1,sizeof(TGAcompare),file)!=sizeof(TGAcompare) || // Are There 12
  memcmp(TGAheader,TGAcompare,sizeof(TGAheader))!=0 || // Does The Header Match
  fread(header,1,sizeof(header),file)!=sizeof(header)) // If So Read Next 6 Header By
{
  fclose(file); // If Anything Failed, Close The File
  return false; // Return False
}
```

If everything went ok, we now have enough information to define some important variables. The first variable we want to define is **width**. We want **width** to equal the width of the TGA file. We can find out the TGA width by multiplying the value stored in **header[1]** by 256. We then add the lowbyte which is stored in **header[0]**.

The height is calculated the same way but instead of using the values stored in **header[0]** and **header[1]** we use the values stored in **header[2]** and **header[3]**.

After we have calculated the **width** and **height** we check to see if either the **width** or **height** is less than or equal to 0. If either of the two variables is less than or equal to zero, the file will be closed, and false will be returned.

We also check to see if the TGA is a 24 or 32 bit image. We do this by checking the value stored at **header[4]**. If the value is not 24 or 32 (bit), the file will be closed, and false will be returned.

In case you have not realized. A return of false will cause the program to fail with the message "Initialization Failed". Make sure your TGA is an uncompressed 24 or 32 bit image!

```
texture->width = header[1] * 256 + header[0]; // Determine The TGA Width (highbyte
texture->height = header[3] * 256 + header[2]; // Determine The TGA Height (highbyte
if( texture->width <=0 || // Is The Width Less Than Or Equal To Zero
texture->height <=0 || // Is The Height Less Than Or Equal To Zero</pre>
```

```
(header[4]!=24 && header[4]!=32)) // Is The TGA 24 or 32 Bit?
{
  fclose(file); // If Anything Failed, Close The File
  return false; // Return False
}
```

Now that we have calculated the image **width** and **height** we need to calculate the bits per pixel, bytes per pixel and image size.

The value in header[4] is the bits per pixel. So we set bpp to equal header[4].

If you know anything about bits and bytes, you know that 8 bits makes a byte. To figure out how many bytes per pixel the TGA uses, all we have to do is divide bits per pixel by 8. If the image is 32 bit, **bytesPerPixel** will equal 4. If the image is 24 bit, **bytesPerPixel** will equal 3.

To calculate the image size, we multiply **width** * **height** * **bytesPerPixel**. The result is stored in **imageSize**. If the image was 100x100x32 bit our image size would be 100 * 100 * 32/8 which equals 10000 * 4 or 40000 bytes!

```
texture->bpp = header[4]; // Grab The TGA's Bits Per Pixel (24 or 32)
bytesPerPixel = texture->bpp/8; // Divide By 8 To Get The Bytes Per Pixel
imageSize = texture->width*texture->height*bytesPerPixel; // Calculate The Memory Re
```

Now that we know how many bytes our image is going to take, we need to allocate some memory. The first line below does the trick. **imageData** will point to a section of ram big enough to hold our image. malloc(**imagesize**) allocates the memory (sets memory aside for us to use) based on the amount of ram we request (**imageSize**).

The "if" statement has a few tasks. First it checks to see if the memory was allocated properly. If not, **imageData** will equal NULL, the file will be closed, and false will be returned.

If the memory was allocated, we attempt to read the image data from the file into the allocated memory. The line fread(texture->imageData, 1, imageSize, file) does the trick. fread means file read. imageData points to the memory we want to store the data in. 1 is the size of data we want to read in bytes (we want to read 1 byte at a time). imageSize is the total number of bytes we want to read. Because imageSize is equal to the total amount of ram required to hold the image, we end up reading in the entire image. file is the handle for our open file.

After reading in the data, we check to see if the amount of data we read in is the same as the value stored in **imageSize**. If the amount of data read and the value of **imageSize** is not the same, something went wrong. If any data was loaded, we will free it. (release the memory we allocated). The file will be closed, and false will be returned.

```
texture->imageData=(GLubyte *)malloc(imageSize); // Reserve Memory To Hold The TGA Dat
if( texture->imageData==NULL || // Does The Storage Memory Exist?
  fread(texture->imageData, 1, imageSize, file)!=imageSize) // Does The Image Size M:
{
  if(texture->imageData!=NULL) // Was Image Data Loaded
    free(texture->imageData); // If So, Release The Image Data
  fclose(file); // Close The File
  return false; // Return False
}
```

If the data was loaded properly, things are going good :) All we have to do now is swap the Red and Blue bytes. In OpenGL we use RGB (red, green, blue). The data in a TGA file is stored BGR (blue, green, red). If we didn't swap the red and blue bytes, anything in the picture that should be red would be blue and anything that should be blue would be red.

Jeff Molofee's OpenGL Windows Tutorial #25

The first thing we do is create a loop (i) that goes from 0 to **imageSize**. By doing this, we can loop through all of the image data. Our loop will increase by steps of 3 (0, 3, 6, 9, etc) if the TGA file is 24 bit, and 4 (0, 4, 8, 12, etc) if the image is 32 bit. The reason we increase by steps is so that the value at i is always going to be the first byte ([b]lue byte) in our group of 3 or 4 bytes.

Inside the loop, we store the [b]lue byte in our **temp** variable. We then grab the red byte which is stored at **texture->imageData[i+2]** (Remember that TGAs store the colors as BGR[A]. B is i+0, G is i+1 and R is i+2) and store it where the [b]lue byte used to be.

Lastly we move the [b]lue byte that we stored in the **temp** variable to the location where the [r]ed byte used to be (i+2), and we close the file with fclose(file).

If everything went ok, the TGA should now be stored in memory as usable OpenGL texture data!

```
for(GLuint i=0; i<int(imageSize); i+=bytesPerPixel) // Loop Through The Image Data
{    // Swaps The 1st And 3rd Bytes ('R'ed and 'B'lue)
    temp=texture->imageData[i]; // Temporarily Store The Value At Image Data 'i'
    texture->imageData[i] = texture->imageData[i + 2]; // Set The 1st Byte To The Value
    texture->imageData[i + 2] = temp; // Set The 3rd Byte To The Value In 'temp' (1st F
}
fclose (file); // Close The File
```

Now that we have usable data, it's time to make a texture from it. We start off by telling OpenGL we want to create a texture in the memory pointed to by **&texture[0].texID**.

It's important that you understand a few things before we go on. In the InitGL() code, when we call LoadTGA() we pass it two parameters. The first parameter is **&textures[0]**. In LoadTGA() we don't make reference to **&textures[0]**. We make reference to **&texture[0]** (no 's' at the end). When we modify **&texture[0]** we are actually modifying **textures[0]**. **texture[0]** assumes the identity of **textures[0]**. I hope that makes sense.

So if we wanted to create a second texture, we would pass the parameter **&textures[1]**. In LoadTGA() any time we modified **texture[0]** we would be modifying **textures[1]**. If we passed **&textures[2]**, **texture[0]** would assume the identity of **&textures[2]**, etc.

Hard to explain, easy to understand. Of course I wont be happy until I make it really clear :) Last example in english using an example. Say I had a box. I called it box #10. I gave it to my friend and asked him to fill it up. My friend could care less what number it is. To him it's just a box. So he fills what he calls "just a box". He gives it back to me. To me he just filled Box #10 for me. To him he just filled a box. If I give him another box called box #11 and say hey, can you fill this. He'll again think of it as just "box". He'll fill it and give it back to me full. To me he's just filled box #11 for me.

When I give LoadTGA **&textures[1]** it thinks of it as **&texture[0]**. It fills it with texture information, and once it's done I am left with a working **textures[1]**. If I give LoadTGA **&textures[2]** it again thinks of it as **&texture[0]**. It fills it with data, and I'm left with a working **textures[2]**. Make sense :)

Anyways... On to the code! We tell LoadTGA() to build our texture. We bind the texture, and tell OpenGL we want it to be linear filtered.

```
// Build A Texture From The Data
glGenTextures(1, &texture[0].texID); // Generate OpenGL texture IDs
glBindTexture(GL_TEXTURE_2D, texture[0].texID); // Bind Our Texture
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Linear Filtered
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Linear Filtered
```

Now we check to see if the TGA file was 24 or 32 bit. If the TGA was 24 bit, we set the **type** to GL_RGB. (no alpha channel). If we didn't do this, OpenGL would try to build a texture with an alpha

}

channel. The alpha information wouldn't be there, and the program would probably crash or give an error message.

```
if (texture[0].bpp==24) // Was The TGA 24 Bits
{
   type=GL_RGB; // If So Set The 'type' To GL_RGB
}
```

Now we build our texture, the same way we've always done it. But instead of putting the type in ourselves (GL_RGB or GL_RGBA), we substitute the variable **type**. That way if the program detected that the TGA was 24 bit, the type will be GL_RGB. If our program detected that the TGA was 32 bit, the type would be GL_RGBA.

After the texture has been built, we return true. This lets the InitGL() code know that everything went ok.

```
glTexImage2D(GL_TEXTURE_2D, 0, type, texture[0].width, texture[0].height, 0, type, GL_UI
return true; // Texture Building Went Ok, Return True
```

The code below is our standard build a font from a texture code. You've all seen this code before if you've gone through all the tutorials up until now. Nothing really new here, but I figured I'd include the code to make following through the program a little easier.

Only real difference is that I bind to **textures[0].texID**. Which points to the font texture. Only real difference is that **.texID** has been added.

```
GLvoid BuildFont(GLvoid) // Build Our Font Display List
   base=glGenLists(256); // Creating 256 Display Lists
   glBindTexture(GL_TEXTURE_2D, textures[0].texID); // Select Our Font Texture
   for (int loop1=0; loop1<256; loop1++)</pre>
                                           // Loop Through All 256 Lists
   {
      float cx=float(loop1%16)/16.0f; // X Position Of Current Character
float cy=float(loop1/16)/16.0f; // Y Position Of Current Character
      glNewList(base+loop1,GL_COMPILE); // Start Building A List
         glBegin(GL_QUADS); // Use A Quad For Each Character
            glTexCoord2f(cx, 1.0f-cy-0.0625f);
                                                  // Texture Coord (Bottom Left)
            glVertex2d(0,16); // Vertex Coord (Bottom Left)
            glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f); // Texture Coord (Bottom Right)
            glVertex2i(16,16); // Vertex Coord (Bottom Right)
            glTexCoord2f(cx+0.0625f,1.0f-cy-0.001f);
                                                       // Texture Coord (Top Right)
            glVertex2i(16,0); // Vertex Coord (Top Right)
            glTexCoord2f(cx,1.0f-cy-0.001f);
                                                // Texture Coord (Top Left)
            glVertex2i(0,0); // Vertex Coord (Top Left)
         glEnd(); // Done Building Our Quad (Character)
         glTranslated(14,0,0); // Move To The Right Of The Character
      glEndList(); // Done Building The Display List
       // Loop Until All 256 Are Built
   }
}
```

KillFont is still the same. We created 256 display lists, so we need to destroy 256 display lists when the program closes.

```
GLvoid KillFont(GLvoid) // Delete The Font From Memory
{
   glDeleteLists(base,256); // Delete All 256 Display Lists
}
```

The glPrint() code has only changed a bit. The letters are all stretched on the y axis. Making the letters very tall. I've explained the rest of the code in other tutorials. The stretching is accomplished with the glScalef(x,y,z) command. We leave the ratio at 1.0 on the x axis, we double the size on the y axis (2.0), and we leave it at 1.0 on the z axis.

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...) // Where The Printing Hag
  char text[1024]; // Holds Our String
  va_list ap; // Pointer To List Of Arguments
  if (fmt == NULL) // If There's No Text
     return; // Do Nothing
  va_start(ap, fmt); // Parses The String For Variables
     vsprintf(text, fmt, ap); // And Converts Symbols To Actual Numbers
  va_end(ap); // Results Are Stored In Text
   if (set>1) // Did User Choose An Invalid Character Set?
   {
     set=1; // If So, Select Set 1 (Italic)
  }
  glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
  glLoadIdentity(); // Reset The Modelview Matrix
  glTranslated(x,y,0); // Position The Text (0,0 - Top Left)
  glListBase(base-32+(128*set)); // Choose The Font Set (0 or 1)
  glScalef(1.0f,2.0f,1.0f); // Make The Text 2X Taller
  glCallLists(strlen(text),GL_UNSIGNED_BYTE, text); // Write The Text To The Screen
  glDisable(GL_TEXTURE_2D); // Disable Texture Mapping
}
```

ReSizeGLScene() sets up an ortho view. Nothing really new. 0,1 is the top left of the screen. 639,480 is the bottom right. This gives us exact screen coordinates in 640 x 480 resolution. I'm not sure why the screen starts at zero on the x axis, but it does :)

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Wind(
{
    if (height==0) // Prevent A Divide By Zero By
    {
        height=1; // Making Height Equal One
    }
    glViewport(0,0,width,height); // Reset The Current Viewport
    glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
    glLoadIdentity(); // Reset The Projection Matrix
    glOrtho(0.0f,640,480,0.0f,-1.0f,1.0f); // Create Ortho 640x480 View (0,0 At Top Left)
    glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix
    glLoadIdentity(); // Reset The Modelview Matrix
}
```

The init code is very minimal. We load our TGA file. Notice that the first parameter passed is **&textures[0]**. The second parameter is the name of the file we want to load. In this case, we want to load the Font.TGA file. If LoadTGA() returns false for any reason, the if statement will also return false, causing the program to quit with an "initialization failed" message.

If you wanted to load a second texture you could use the following code: if ((!LoadTGA(&textures [0],"image1.tga")) && (!LoadTGA(&textures[1],"image2.tga"))) { }

After we load the TGA (creating our texture), we build our font, set shading to smooth, set the background color to black, enable clearing of the depth buffer, and select our font texture (bind to it).

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
    if (!LoadTGA(&textures[0],"Data/Font.TGA")) // Load The Font Texture
    {
        return false; // If Loading Failed, Return False
    }
    BuildFont(); // Build The Font
    glShadeModel(GL_SMOOTH); // Enable Smooth Shading
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Black Background
    glClearDepth(1.0f); // Depth Buffer Setup
    glBindTexture(GL_TEXTURE_2D, textures[0].texID); // Select Our Font Texture
```

Now for something new. A wonderful GL command called glScissor(x,y,w,h). What this command does is creates almost what you would call a window. When GL_SCISSOR_TEST is enabled, the only portion of the screen that you can alter is the portion inside the scissor window. The command below creates a window starting at 1 on the x axis, and 64 pixels up from the bottom of the screen on the y axis. The scissor window will be 638 pixels wide, and 288 pixels tall.

It's important to note that OpenGL assumes the first two numbers represent the lower left corner of the scissor box. With that in mind, 64 represents 64 pixels from the bottom of the screen, not the top.

This means the bottom left of the scissor window will be at 1,416 (480-64), and the top right of the scissor window will be at 638,128 (416-288).

We start off with scissor testing disabled, meaning we can draw anywhere we want on the screen. Once scissor testing has been enabled. Anything we draw OUTSIDE the scissor window will not show up. You could draw a HUGE quad on the screen from 0,0 to 639,480, and you would only see the quad inside the scissor windows, the rest of the screen would be unaffected. Very nice command indeed.

Last thing we do is return true so that our program knows that initialization went ok.

```
glScissor(1,64,637,288); // Define Scissor Region
return TRUE; // Initialization Went OK
```

}

The draw code is completely new :) we start off by creating a variable of type char called **token**. Token will hold parsed text later on in the code.

We have another variable called **cnt**. I use this variable both for counting the number of extensions supported, and for positioning the text on the screen. **cnt** is reset to zero every time we call DrawGLScene.

We clear the screen and depth buffer and then set the color to bright red (full red intensity, 50% green, 50% blue). at 50 on the x axis and 16 on the y axis we write teh word "Renderer". We also write "Vendor" and "Version" at the top of the screen. The reason each word does not start at 50 on the x axis is because I right justify the words (they all line up on the right side).

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
    char *token; // Storage For Our Token
    int cnt=0; // Local Counter Variable
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
    glColor3f(1.0f,0.5f,0.5f); // Set Color To Bright Red
    glPrint(50,16,1,"Renderer"); // Display Renderer
    glPrint(80,48,1,"Vendor"); // Display Vendor Name
    glPrint(66,80,1,"Version"); // Display Version
```

Now that we have text on the screen, we change the color orange, and grab the renderer, vendor name and version number from the video card. We do this by passing GL_RENDERER, GL_VENDOR & GL_VERSION to glGetString(). glGetString will return the requested renderer name, vendor name and version number. The information returned will be text so we need to cast the return information from glGetString as char. All this means is that we tell the program we want the information returned to be characters (text). If you don't include the (char *) you will get an error message. We're printing text, so we need text returned. We grab all three pieces of information and write the information we've grabbed to the right of the previous text.

The information we get from glGetString(GL_RENDERER) will be written beside the red text "Renderer", the information we get from glGetString(GL_VENDOR) will be written to the right of "Vendor", etc.

I'd like to explain casting in more detail, but I'm not really sure of a good way to explain it. If anyone has a good explanation, send it in, and I'll modify my explanation.

After we have the renderer information, vendor information and version number written to the screen, we change the color to a bright blue, and write "NeHe Productions" at the bottom of the screen :) Of course you can change this to anything you want.

```
glColor3f(1.0f,0.7f,0.4f); // Set Color To Orange
glPrint(200,16,1,(char *)glGetString(GL_RENDERER)); // Display Renderer
glPrint(200,48,1,(char *)glGetString(GL_VENDOR)); // Display Vendor Name
glPrint(200,80,1,(char *)glGetString(GL_VERSION)); // Display Version
glColor3f(0.5f,0.5f,1.0f); // Set Color To Bright Blue
glPrint(192,432,1,"NeHe Productions"); // Write NeHe Productions At The Bottom Of The
```

Now we draw a nice white border around the screen, and around the text. We start off by resetting the modelview matrix. Because we've been printing text to the screen, and we might not be at 0,0 on the screen, it's a safe thing to do.

We then set the color to white, and start drawing our borders. A line strip is actually pretty easy to use. You tell OpenGL you want to draw a line strip with glBegin(GL_LINE_STRIP). Then we set the first vertex. Our first vertex will be on the far right side of the screen, and about 63 pixels up from the bottom of the screen (639 on the x axis, 417 on the y axis). Then we set the second vertex. We stay at the same location on the y axis (417), but we move to the far left side of the screen on the x axis (0). A line will be drawn from the right side of the screen (639,417) to the left side of the screen (0,417).

You need to have at least two vertices in order to draw a line (common sense). From the left side of the screen, we move down, right, and then straight up (128 on the y axis).

We then start another line strip, and draw a second box at the top of the screen. If you need to draw ALOT of connected lines, line strips can definitely cut down on the amount of code required as opposed to using regular lines (GL_LINES).

glLoadIdentity(); // Reset The ModelView Matrix

```
glColor3f(1.0f,1.0f,1.0f); // Set The Color To White
glBegin(GL_LINE_STRIP); // Start Drawing Line Strips (Something New)
glVertex2d(639,417); // Top Right Of Bottom Box
glVertex2d( 0,417); // Top Left Of Bottom Box
glVertex2d( 0,480); // Lower Left Of Bottom Box
glVertex2d(639,480); // Lower Right Of Bottom Box
glVertex2d(639,128); // Up To Bottom Right Of Top Box
glEnd(); // Done First Line Strip
glBegin(GL_LINE_STRIP); // Start Drawing Another Line Strip
glVertex2d(639,128); // Bottom Left Of Top Box
glVertex2d(639,128); // Bottom Right Of Top Box
glVertex2d(639,128); // Bottom Right Of Top Box
glVertex2d(639,11; // Top Right Of Top Box
glVertex2d( 0, 1); // Top Left Of Top Box
glVertex2d( 0, 417); // Down To Top Left Of Bottom Box
```

Now for the fun stuff! We enable scissor testing with glEnable(GL_SCISSOR_TEST). Once scissor testing is enabled we can't draw outside the scissor region that we defined in InitGL().

The second line of code below creates a variable called **text** that will hold the characters returned by glGetString(GL_EXTENSIONS). malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1) allocates enough memory to hold the entire string returned +1 (so if the string was 50 characters, **text** would be able to hold all 50 characters).

The next line copies the GL_EXTENSIONS information to **text**. If we modify the GL_EXTENSIONS information directly, big problems will occur, so instead we copy the information into **text**, and then manipulate the information stored in **text**. Basically we're just taking a copy, and storing it in the variable **text**.

glEnable(GL_SCISSOR_TEST); // Enable Scissor Testing

```
char* text=(char*)malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1); // Allocate Me
strcpy (text,(char *)glGetString(GL_EXTENSIONS)); // Grab The Extension List, Store II
```

Now for something new. Lets pretend that after grabbing the extension information from the video card, the variable **text** had the following string of text stored in it... "GL_ARB_multitexture GL_EXT_abgr GL_EXT_bgra". strtok(TextToAnalyze,TextToFind) will scan through the variable **text** until it finds a " " (space). Once it finds a space, it will copy the text UP TO the space into the variable **token**. So in our little example, **token** would be equal to "GL_ARB_multitexture". The space is then replaced with a marker. More about this in a minute.

Next we create a loop that stops once there is no more information left in **text**. If there is no information in **text**, **token** will be equal to nothing (NULL) and the loop will stop.

We increase the counter variable (cnt) by one, and then check to see if the value in cnt is higher than the value of maxtokens. If cnt is higher than maxtokens we make maxtokens equal to cnt. That way if the counter hits 20, maxtokens will also equal 20. It's an easy way to keep track of the maximum value of cnt.

```
token=strtok(text," "); // Parse 'text' For Words, Seperated By " " (spaces)
while(token!=NULL) // While The Token Isn't NULL
{
    cnt++; // Increase The Counter
    if (cnt>maxtokens) // Is 'maxtokens' Less Than 'cnt'
    {
        maxtokens=cnt; // If So, Set 'maxtokens' Equal To 'cnt'
    }
```

So we have stored the first extension from our list of extensions in the variable token. Next thing to

do is set the color to bright green. We then print the variable **cnt** on the left side of the screen. Notice that we print at 0 on the x axis. This should erase the left (white) border that we drew, but because scissor testing is on, pixels drawn at 0 on the x axis wont be modified. The border can't be drawn over.

The variable is drawn on the far left side of the screen (0 on the x axis). We start drawing at 96 on the y axis. To keep all the text from drawing to the same spot on the screen, we add ($cnt^{*}32$) to 96. So if we are displaying the first extension, cnt will equal 1, and the text will be drawn at 96+(32*1) (128) on the y axis. If we display the second extension, cnt will equal 2, and the text will be drawn at 96+(32*2) (160) on the y axis.

Notice I also subtract **scroll**. When the program first runs, **scroll** will be equal to 0. So our first line of text is drawn at 96+(32*1)-0. If you press the DOWN ARROW, **scroll** is increased by 2. If **scroll** was 4, the text would be drawn at 96+(32*1)-4. That means the text would be drawn at 124 instead of 128 on the y axis because of **scroll** being equal to 4. The top of our scissor window ends at 128 on the y axis. Any part of the text drawn from lines 124-127 on the y axis will not appear on the screen.

Same thing with the bottom of the screen. If **cnt** was equal to 11 and **scroll** was equal to 0, the text would be drawn at 96+(32*11)-0 which is 448 on the y axis. Because the scissor window only allows us to draw as far as line 416 on the y axis, the text wouldn't show up at all.

The final result is that we end up with a scrollable window that only allows us to look at 288/32 (9) lines of text. 288 is the height of our scissor window. 32 is the height of the text. By changing the value of **scroll** we can move the text up or down (offset the text).

The effect is similar to a movie projector. The film rolls by the lens, and all you see is the current frame. You don't see the frame above or below. The lens acts as a window similar to the window created by the scissor test.

After we have drawn the current count (**cnt**) to the screen, we change the color to yellow, move 50 pixels to the right on the x axis, and we write the text stored in the variable **token** to the screen.

Using our example above, the first line of text displayed on the screen should look like this:

1 GL_ARB_multitexture

```
glColor3f(0.5f,1.0f,0.5f); // Set Color To Bright Green
glPrint(0,96+(cnt*32)-scroll,0,"%i",cnt); // Print Current Extension Number
```

After we have drawn the current count to the screen, we change the color to yellow, move 50 pixels to the right on the x axis, and we write the text stored in the variable **token** to the screen.

Using our example above, the first line of text displayed on the screen should look like this:

1 GL_ARB_multitexture

```
glColor3f(1.0f,1.0f,0.5f); // Set Color To Yellow
glPrint(50,96+(cnt*32)-scroll,0,token); // Print The Current Token (Parsed Extensic
```

After we have displayed the value of **token** on the screen, we need to check through the variable **text** to see if any more extensions are supported. Instead of using **token**=strtok(**text**," ") like we did above, we replace **text** with NULL. This tells the command strtok to search from the last marker to the NEXT space in the string of text (**text**).

In our example above ("GL_ARB_multitexturemarkerGL_EXT_abgr GL_EXT_bgra") there will now be a marker after the text "GL_ARB_multitexture". The line below will start search FROM the marker to the next space. Everything from the marker to the next space will be stored in **token**. **token** should end up being "GL_EXT_abgr", and **text** will end up being "GL_ARB_multitexturemarkerGL_EXT_abgrmarkerGL_EXT_bgra".

Once strtok() has run out of text to store in token, token will become NULL and the loop will stop.

```
token=strtok(NULL," "); // Search For The Next Token
}
```

After all of the extensions have been parsed from the variable **text** we can disable scissor testing, and free the variable **text**. This releases the ram we were using to hold the information we got from glGetString(GL_EXTENSIONS).

The next time DrawGLScene() is called, new memory will be allocated. A fresh copy of the information returned by glGetStrings(GL_EXTENSIONS) will be copied into the variable **text** and the entire process will start over.

```
glDisable(GL_SCISSOR_TEST); // Disable Scissor Testing
```

free (text); // Free Allocated Memory

The first line below isn't necessary, but I thought it might be a good idea to talk about it, just so everyone knows that it exists. The command glFlush() basically tells OpenGL to finish up what it's doing. If you ever notice flickering in your program (quads disappearing, etc). Try adding the flush command to the end of DrawGLScene. It flushes out the rendering pipeline. You may notice flickering if you're program doesn't have enough time to finish rendering the scene.

Last thing we do is return true to show that everything went ok.

```
glFlush(); // Flush The Rendering Pipeline
return TRUE; // Everything Went OK
```

}

The only thing to note in KillGLWindow() is that I have added KillFont() at the end. That way whenever the window is killed, the font is also killed.

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
   if (fullscreen) // Are We In Fullscreen Mode?
   {
     ChangeDisplaySettings(NULL,0);
                                     // If So Switch Back To The Desktop
      ShowCursor(TRUE); // Show Mouse Pointer
   }
  if (hRC) // Do We Have A Rendering Context?
   ł
      if (!wglMakeCurrent(NULL,NULL)) // Are We Able To Release The DC And RC Contexts?
      {
        MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR", MB_OK | MB_ICONINI
      }
      if (!wqlDeleteContext(hRC)) // Are We Able To Delete The RC?
      {
        MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN ERROR", MB_OK | MB_I(
     hRC=NULL; // Set RC To NULL
   }
```

```
if (hDC && !ReleaseDC(hWnd,hDC)) // Are We Able To Release The DC
{
    MessageBox(NULL, "Release Device Context Failed.", "SHUTDOWN ERROR", MB_OK | MB_ICONINF(
    hDC=NULL; // Set DC To NULL
}
if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?
{
    MessageBox(NULL, "Could Not Release hWnd.", "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION
    hWnd=NULL; // Set hWnd To NULL
}
if (!UnregisterClass("OpenGL", hInstance)) // Are We Able To Unregister Class
{
    MessageBox(NULL, "Could Not Unregister Class.", "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION
    hInstance=NULL; // Set hInstance To NULL
}
KillFont(); // Kill The Font
```

CreateGLWindow(), and WndProc() are the same.

}

The first change in WinMain() is the title that appears at the top of the window. It should now read "NeHe's Extensions, Scissoring, Token & TGA Loading Tutorial"

```
HINSTANCE hInstance,
int WINAPI WinMain(
                                            // Instance
                  hPrevInstance, // Previous Instance
        HINSTANCE
                  lpCmdLine, // Command Line Parameters
        LPSTR
        int
                nCmdShow) // Window Show State
       msg; // Windows Message Structure
  MSG
        done=FALSE; // Bool Variable To Exit Loop
  BOOL
  // Ask The User Which Screen Mode They Prefer
  if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreen?", MB_
  {
     fullscreen=FALSE; // Windowed Mode
  }
  // Create Our OpenGL Window
  if (!CreateGLWindow("NeHe's Token, Extensions, Scissoring & TGA Loading Tutorial",640,48
  {
     return 0; // Quit If Window Was Not Created
  }
               // Loop That Runs While done=FALSE
  while(!done)
  {
     if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
        if (msg.message==WM_QUIT) // Have We Received A Quit Message?
        {
           done=TRUE; // If So done=TRUE
        else // If Not, Deal With Window Messages
           DispatchMessage(&msg); // Dispatch The Message
        }
           // If There Are No Messages
     else
  // Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
        if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit
        {
           done=TRUE; // ESC or DrawGLScene Signalled A Quit
        }
        else // Not Time To Quit, Update Screen
```

```
{
    SwapBuffers(hDC); // Swap Buffers (Double Buffering)
    if (keys[VK_F1]) // Is F1 Being Pressed?
    {
        keys[VK_F1]=FALSE; // If So Make Key FALSE
        KillGLWindow(); // Kill Our Current Window
        fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
        if (!CreateGLWindow("NeHe's Token, Extensions, Scissoring & TGA Loading Tutc
        {
            return 0; // Quit If Window Was Not Created
        }
     }
}
```

The code below checks to see if the up arrow is being pressed if it is, and **scroll** is greater than 0, we decrease **scroll** by 2. This causes the text to move down the screen.

```
if (keys[VK_UP] && (scroll>0)) // Is Up Arrow Being Pressed?
{
    scroll-=2; // If So, Decrease 'scroll' Moving Screen Down
}
```

If the down arrow is being pressed and **scroll** is less than (32*(**maxtokens**-9)) **scroll** will be increased by 2, andd the text on the screen will scroll upwards.

32 is the number of lines that each letter takes up. Maxtokens is the total amount of extensions that your video card supports. We subtract 9, because 9 lines can be shown on the screen at once. If we did not subtract 9, we could scroll past the end of the list, causing the list to scroll completely off the screen. Try leaving the -9 out if you're not sure what I mean.

I hope that you found this tutorial interesting. By the end of this tutorial you should know how to read the vendor name, renderer and version number from your video card. You should also know how to find out what extensions are supported on any video card that supports OpenGL. You should know what scissor testing is, and how it can be used in OpenGL projects of your own, and lastly, you should know how to load TGA Images instead of Bitmap Images for use as textures.

If you find any problems with the tutorial, or you find the information to hard to understand, let me know. I want the tutorials to be the best they can be. Your feedback is important!

Jeff Molofee (NeHe)

}

* DOWNLOAD Visual C++ Code For This Lesson.

Lesson 26

Welcome to yet another exciting tutorial! This time we will focus on the effect rather than the graphics, although the final result is pretty cool looking! In this tutorial you will learn how to morph seamlessly from one object to another. Similar to the effect I use in the dolphin demo. Although there are a few catches. First thing to note is that each object must have the same amount of points. Very rare to luck out and get 3 object made up of exactly the same amount of vertices, but it just so happens, in this tutorial we have 3 objects with exactly the same amount of points :) Don't get me wrong, you can use objects with different values, but the transition from one object to another is odd looking and not as smooth.

You will also learn how to read object data from a file. Similar to the format used in lesson 10, although it shouldn't be hard to modify the code to read .ASC files or some other text type data files. In general, it's a really cool effect, a really cool tutorial, so lets begin!

We start off as usual. Including all the required header files, along with the math and standard input / output headers. Notice we don't include glaux. That's because we'll be drawing points rather than textures in this tutorial. After you've got the tutorial figured out, you can try playing with Polygons, Lines, and Textures!

	h> // Heade // Math Library H	r File For Windows
#include <stdio.h< td=""><td>> // Heade</td><td>r File For Standard Input/Output</td></stdio.h<>	> // Heade	r File For Standard Input/Output
<pre>#include <gl\gl.h< pre=""></gl\gl.h<></pre>		r File For The OpenGL32 Library
#include <gl\glu.< td=""><td>h> // Heade</td><td>r File For The GLu32 Library</td></gl\glu.<>	h> // Heade	r File For The GLu32 Library
HDC	hDC=NULL;	// Device Context Handle
HGLRC	hRC=NULL;	// Rendering Context Handle
HWND	hWnd=NULL;	// Window Handle
HINSTANCE	hInstance;	// Instance Handle
bool	keys[256];	// Key Array
bool	active=TRUE;	// Program's Active
bool	fullscreen=TRUE;	// Default Fullscreen To True

After setting up all the standard variables, we will add some new variables. **xrot**, **yrot** and **zrot** will hold the current rotation values for the x, y and z axes of the onscreen object. **xspeed**, **yspeed** and **zspeed** will control how fast the object is rotating on each axis. **cx**, **cy** and **cz** control the position of the object on the screen (where it's drawn left to right **cx**, up and down **cy** and into and out of the screen **cz**)

The variable **key** is a variable that I have included to make sure the user doesn't try to morph from the first shape back into the first shape. This would be pretty pointless and would cause a delay while the points were trying to morph to the position they're already in.

step is a counter variable that counts through all the steps specified by **steps**. If you increase the value of **steps** it will take longer for the object to morph, but the movement of the points as they morph will be smoother. Once **step** is equal to **steps** we know the morphing has been completed.

The last variable **morph** lets our program know if it should be morphing the points or leaving them where they are. If it's TRUE, the object is in the process of morphing from one shape to another.

GLfloat	<pre>xrot,yrot,zrot, // X, Y & Z Rotation xspeed,yspeed,zspeed, // X, Y & Z Spin Speed cx,cy,cz=-15; // X, Y & Z Position</pre>
int	<pre>key=1; // Used To Make Sure Same Morph Key Is Not Pressed</pre>
int	step=0,steps=200; // Step Counter And Maximum Number Of Steps
bool	morph=FALSE; // Default morph To False (Not Morphing)

Now we create a structure to keep track of a vertex. The structure will hold the x, y and z values of any point on the screen. The variables x, y & z are all floating point so we can position the point anywhere on the screen with great accuracy. The structure name is **VERTEX**.

typedef struct {	//	Structure For 3D Points
float } VERTEX;		y, z; // X, Y & Z Points Called VERTEX

We already have a structure to keep track of vertices, and we know that an object is made up of many vertices so lets create an **OBJECT** structure. The first variable **verts** is an integer value that will hold the number of vertices required to make up an object. So if our object has 5 points, the value of **verts** will be equal to 5. We will set the value later in the code. For now, all you need to know is that **verts** keeps track of how many points we use to create the object.

The variable **points** will reference a single VERTEX (x, y and z values). This allows us to grab the x, y or z value of any point using **points[{point we want to access}].{x, y or z}.**

The name of this structure is... you guessed it... OBJECT!

typedef struct	// Structure For An Object
{ int	verts; // Number Of Vertices For The Object
VERTEX	*points; // One Vertice (Vertex x,y & z)
} OBJECT;	// Called OBJECT

Now that we have created a **VERTEX** structure and an **OBJECT** structure we can define some objects.

The variable **maxver** will be used to keep track of the maximum number of variables used in any of the objects. If one object only had 5 points, another had 20, and the last object had 15, the value of **maxver** would be equal to the greatest number of points used. So **maxver** would be equal to 20.

After we define **maxver** we can define the objects. **morph1**, **morph2**, **morph3**, **morph4** & **helper** are all defined as an **OBJECT**. ***sour** & ***dest** are defined as OBJECT* (pointer to an object). The object is made up of verticies (**VERTEX**). The first 4 **morph{num}** objects will hold the 4 objects we want to morph to and from. **helper** will be used to keep track of changes as the object is morphed. ***sour** will point to the source object and ***dest** will point to the object we want to morph to (destination object).

int	maxver; // Will Eventually Hold	The Maximum Number Of Vertices
OBJECT	<pre>morph1,morph2,morph3,morph4,</pre>	<pre>// Our 4 Morphable Objects (morph1,2,</pre>
	helper,*sour,*dest;	// Helper Object, Source Obj

Same as always, we declare WndProc().

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration
```

The code below allocates memory for each object, based on the number of vertices we pass to n. *k will point to the object we want to allocate memory for.

The line inside the $\{ \}$'s allocates the memory for object **k**'s points. A point is an entire VERTEX (3) floats). The memory allocated is the size of VERTEX (3 floats) multiplied by the number of points (n). So if there were 10 points (n=10) we would be allocating room for 30 floating point values (3 floats * 10 points).

```
void objallocate(OBJECT *k,int n) // Allocate Memory For Each Object
        // And Defines points
{
        k->points=(VERTEX*)malloc(sizeof(VERTEX)*n); // Sets points Equal To VERTEX * Numb
}
        // (3 Points For Each Vertice)
```

The following code frees the object, releasing the memory used to create the object. The object is passed as k. The free command tells our program to release all the points used to make up our obiect (k).

```
void objfree(OBJECT *k) // Frees The Object (Releasing The Memory)
{
        free(k->points); // Frees Points
}
```

The code below reads a string of text from a file. The pointer to our file structure is passed to *f. The variable string will hold the text that we have read in.

We start off be creating a do / while loop. fgets() will read up to 255 characters from our file f and store the characters at *string. If the line read is blank (carriage return \n), the loop will start over, attempting to find a line with text. The while() statement checks for blank lines and if found starts over again.

After the string has been read in we return.

```
void readstr(FILE *f, char *string) // Reads A String From File (f)
{
                 // Do This
        do
                 fgets(string, 255, f); // Gets A String Of 255 Chars Max From f (File
        } while ((string[0] == '/') || (string[0] == '\n')); // Until End Of Line Is Reac
        return; // Return
}
```

Now we load in an object. *name points to the filename. *k points to the object we wish to load data into.

We start off with an integer variable called ver. ver will hold the number of vertices used to build the object.

The variables **rx**, **ry & rz** will hold the x, y & z values of each vertex.

The variable filein is the pointer to our file structure, and oneline[] will be used to hold 255 characters of text.

We open the file **name** for read in text translated mode (meaning CTRL-Z represents the end of a

line). Then we read in a line of text using readstr(filein,oneline). The line of text will be stored in **oneline**.

After we have read in the text, we scan the line of text (**oneline**) for the phrase "Vertices: {some number}{carriage return}. If the text is found, the number is stored in the variable **ver**. This number is the number of vertices used to create the object. If you look at the object text files, you'll see that the first line of text is: Vertices: {some number}.

After we know how many vertices are used we store the results in the objects **verts** variable. Each object could have a different value if each object had a different number of vertices.

The last thing we do in this section of code is allocate memory for the object. We do this by calling objallocate({object name},{number of verts}).

```
void objload(char *name,OBJECT *k) // Loads Object From File (name)
        int
               ver; // Will Hold Vertice Count
        float rx,ry,rz; // Hold Vertex X, Y & Z Position
        FILE
               *filein; // Filename To Open
        char
               oneline[255]; // Holds One Line Of Text (255 Chars Max)
        filein = fopen(name, "rt");
                                         // Opens The File For Reading Text In Translat
        // CTRL Z Symbolizes End Of File In Translated Mode
        readstr(filein,oneline); // Jumps To Code That Reads One Line Of Text From The F
        sscanf(oneline, "Vertices: %d\n", &ver); // Scans Text For "Vertices: ". Numb
        k->verts=ver; // Sets Objects verts Variable To Equal The Value Of ver
        objallocate(k,ver);
                                // Jumps To Code That Allocates Ram To Hold The Object
```

We know how many vertices the object has. We have allocated memory, now all that is left to do is read in the vertices. We create a loop using the variable **i**. The loop will go through all the vertices.

Next we read in a line of text. This will be the first line of valid text underneath the "Vertices: {some number}" line. What we should end up reading is a line with floating point values for x, y & z.

The line is analyzed with sscanf() and the three floating point values are extracted and stored in **rx**, **ry** and **rz**.

```
for (int i=0;i<ver;i++) // Loops Through The Vertices
{
     readstr(filein,oneline); // Reads In The Next Line Of Text
     sscanf(oneline, "%f %f %f", &rx, &ry, &rz); // Searches For 3 Floating P</pre>
```

The following three lines are hard to explain in plain english if you don't understand structures, etc, but I'll try my best :)

The line k->points[i].x=rx can be broken down like this:

rx is the value on the x axis for one of the points.
points[i].x is the x axis position of point[i].
If i is 0 then were are setting the x axis value of point 1, if i is 1, we are setting the x axis value of point 2, and so on.
points[i] is part of our object (which is represented as k).

So if **i** is equal to 0, what we are saying is: The x axis of point 1 (**point[0].x**) in our object (**k**) equals the x axis value we just read from the file (**rx**).

The other two lines set the y & z axis values for each point in our object.

We loop through all the vertices. If there are not enough vertices, an error might occur, so make sure the text at the beginning of the file "Vertices: {some number}" is actually the number of vertices in the

}

file. Meaning if the top line of the file says "Vertices: 10", there had better be 10 Verticies (x, y and z values)!

After reading in all of the verticies we close the file, and check to see if the variable **ver** is greater than the variable **maxver**. If **ver** is greater than **maxver**, we set **maxver** to equal **ver**. That way if we read in one object and it has 20 verticies, **maxver** will become 20. If we read in another object, and it has 40 verticies, **maxver** will become 40. That way we know how many vertices our largest object has.

```
k->points[i].x = rx; // Sets Objects (k) points.x Value To rx
k->points[i].y = ry; // Sets Objects (k) points.y Value To ry
k->points[i].z = rz; // Sets Objects (k) points.z Value To rz
}
fclose(filein); // Close The File
if(ver>maxver) maxver=ver; // If ver Is Greater Than maxver Set maxver Equal To ve
// Keeps Track Of Highest Number Of Vertices Used
```

The next bit of code may look a little intimidating... it's NOT :) I'll explain it so clearly you'll laugh when you next look at it.

What the code below does is calculates a new position for each point when morphing is enabled. The number of the point to calculate is stored in **i**. The results will be returned in the VERTEX **calculate**.

The first variable we create is a VERTEX called **a**. This will give **a** an x, y and z value.

Lets look at the first line. The x value of the VERTEX a equals the x value of **point[i]** (**point[i].x**) in our SOURCE object minus the x value of **point[i]** (**point[i].x**) in our DESTINATION object divided by **steps**.

So lets plug in some numbers. Lets say our source objects first x value is 40 and our destination objects first x value is 20. We already know that **steps** is equal to 200! So that means that $\mathbf{a.x}=(40-20)/200... \mathbf{a.x}=(20)/200... \mathbf{a.x}=0.1$.

What this means is that in order to move from 40 to 20 in 200 steps, we need to move by 0.1 units each calculation. To prove this calculation, multiply 0.1 by 200, and you get 20. 40-20=20 :)

We do the same thing to calculate how many units to move on both the y axis and the z axis for each point. If you increase the value of **steps** the movements will be even more fine (smooth), but it will take longer to morph from one position to another.

The ReSizeGLScene() code hasn't changed so we'll skip over it.

GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Windo

In the code below we set blending for translucency. This allows us to create neat looking trails when the points are moving.

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
    glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Set The Blending Function For Translucency
    glClearColor(0.0f, 0.0f, 0.0f); // This Will Clear The Background Col
    glClearDepth(1.0); // Enables Clearing Of The Depth Buffer
    glDepthFunc(GL_LESS); // The Type Of Depth Test To Do
    glEnable(GL_DEPTH_TEST); // Enables Depth Testing
    glShadeModel(GL_SMOOTH); // Enables Smooth Color Shading
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective C
```

We set the **maxver** variable to 0 to start off. We haven't read in any objects so we don't know what the maximum amount of vertices will be.

Next well load in 3 objects. The first object is a sphere. The data for the sphere is stored in the file sphere.txt. The data will be loaded into the object named **morph1**. We also load a torus, and a tube into objects **morph2** and **morph3**.

```
maxver=0; // Sets Max Vertices To 0 By Default
objload("data/sphere.txt",&morph1); // Load The First Object Into morph1 From File
objload("data/torus.txt",&morph2); // Load The Second Object Into morph2 From Fil
objload("data/tube.txt",&morph3); // Load The Third Object Into morph3 From File
```

The 4th object isn't read from a file. It's a bunch of dots randomly scattered around the screen. Because we're not reading the data from a file, we have to manually allocate the memory by calling objallocate(&morph4,468). 468 means we want to allocate enough space to hold 468 vertices (the same amount of vertices the other 3 objects have).

After allocating the space, we create a loop that assigns a random x, y and z value to each point. The random value will be a floating point value from +7 to -7. (14000/1000=14... minus 7 gives us a max value of +7... if the random number is 0, we have a minimum value of 0-7 or -7).

```
objallocate(&morph4,486); // Manually Reserver Ram For A 4th 468 Vertice Object (
for(int i=0;i<486;i++) // Loop Through All 468 Vertices
{
    morph4.points[i].x=((float)(rand()%14000)/1000)-7; // morph4 x Point Bemorph4.points[i].y=((float)(rand()%14000)/1000)-7; // morph4 y Point Bemorph4.points[i].z=((float)(rand()%14000)/1000)-7; // morph4 z Point Bemorph4.points[i].z=(float)(rand()%14000)/1000)-7; // morph4 z Point Bemorph4.points[i].z=(float)(rand()%14000)/1000)-7; // morph4.points[i].z=(float)(rand()%14000)/1000)-7; // morph4.points[i].z=(float)(rand()%14000)/1000)-7; // morph4.points[i].z=(float)(rand()%14000)/1000)-7; // morph4.points[i].z=(float)(rand()%14000)/1000)-7; //
```

We then load the sphere.txt as a helper object. We never want to modify the object data in morph {1/2/3/4} directly. We modify the helper data to make it become one of the 4 shapes. Because we start out displaying morph1 (a sphere) we start the helper out as a sphere as well.

After all of the objects are loaded, we set the source and destination objects (sour and dest) to equal morph1, which is the sphere. This way everything starts out as a sphere.

```
objload("data/sphere.txt",&helper); // Load sphere.txt Object Into Helper (Used As
sour=dest=&morph1; // Source & Destination Are Set To Equal First Object (
```

return TRUE; // Initialization Went OK

}

Now for the fun stuff. The actual rendering code :)

We start off normal. Clear the screen, depth buffer and reset the modelview matrix. Then we position the object on the screen using the values stored in **cx**, **cy** and **cz**.

Rotations are done using **xrot**, **yrot** and **zrot**.

The rotation angle is increased based on xpseed, yspeed and zspeed.

Finally 3 temporary variables are created tx, ty and tz, along with a new VERTEX called q.

```
void DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The 1
    glLoadIdentity(); // Reset The View
    glTranslatef(cx,cy,cz); // Translate The The Current Position To Start Drawing
    glRotatef(xrot,1,0,0); // Rotate On The X Axis By xrot
    glRotatef(yrot,0,1,0); // Rotate On The Y Axis By yrot
    glRotatef(zrot,0,0,1); // Rotate On The Z Axis By zrot
    xrot+=xspeed; yrot+=yspeed; zrot+=zspeed; // Increase xrot,yrot & zrot by xspee
    GLfloat tx,ty,tz; // Temp X, Y & Z Variables
    VERTEX q; // Holds Returned Calculated Values For One Vertex
```

Now we draw the points and do our calculations if morphing is enabled. glBegin(GL_POINTS) tells OpenGL that each vertex that we specify will be drawn as a point on the screen.

We create a loop to loop through all the vertices. You could use **maxver**, but because every object has the same number of vertices we'll use **morph1.verts**.

Inside the loop we check to see if **morph** is TRUE. If it is we calculate the movement for the current point (i). **q.x**, **q.y** and **q.z** will hold the results. If **morph** is false, **q.x**, **q.y** and **q.z** will be set to 0 (preventing movement).

the points in the **helper** object are moved based on the results of we got from calculate(i). (remember earlier that we calculated a point would have to move 0.1 unit to make it from 40 to 20 in 200 steps).

We adjust the each points value on the x, y and z axis by subtracting the number of units to move from **helper**.

The new helper point is stored in tx, ty and tz. (t{x/y/z}=helper.points[i].{x/y/z}).

```
glBegin(GL_POINTS); // Begin Drawing Points
for(int i=0;i<morph1.verts;i++) // Loop Through All The Verts Of morp
{    // The Same Amount Of Verts For Simplicity, Could Use maxver Als
    if(morph) q=calculate(i); else q.x=q.y=q.z=0; // If morpl
    helper.points[i].x=q.x; // Subtract q.x Units From helper.poi
    helper.points[i].y=q.y; // Subtract q.z Units From helper.poi
    helper.points[i].z; // Make Temp X Variable Equal To Help
    tz=helper.points[i].z; // Make Temp Z Variable Equal To Help</pre>
```

Now that we have the new position calculated it's time to draw our points. We set the color to a bright bluish color, and the draw the first point with glVertex3f(**tx**,**ty**,**tz**). This draws a point at the newly calculated position.

We then darken the color a little, and move 2 steps in the direction we just calculated instead of one. This moves the point to the newly calculated position, and then moves it again in the same direction. So if it was travelling left at 0.1 units, the next dot would be at 0.2 units. After calculating 2 positions ahead we draw the second point.

Finally we set the color to dark blue, and calculate even further ahead. This time using our example we would move 0.4 units to the left instead of 0.1 or 0.2. The end result is a little tail of particles following as the dots move. With blending, this creates a pretty cool effect!

glEnd() tells OpenGL we are done drawing points.

}

```
glColor3f(0,1,1); // Set Color To A Bright Shade Of Off Blue
glVertex3f(tx,ty,tz); // Draw A Point At The Current Temp V
glColor3f(0,0.5f,1); // Darken Color A Bit
tx-=2*q.x; ty-=2*q.y; ty-=2*q.y; // Calculate Two Positions A
glVertex3f(tx,ty,tz); // Draw A Second Point At The Newly C
glColor3f(0,0,1); // Set Color To A Very Dark Blue
tx-=2*q.x; ty-=2*q.y; ty-=2*q.y; // Calculate Two More Positi-
glVertex3f(tx,ty,tz); // Draw A Third Point At The Second N
} // This Creates A Ghostly Tail As Points Move
glEnd(); // Done Drawing Points
```

The last thing we do is check to see if **morph** is TRUE and **step** is less than **steps** (200). If **step** is less than 200, we increase step by 1.

If **morph** is false or **step** is greater than or equal to **steps** (200), **morph** is set to FALSE, the **sour** (source) object is set to equal the **dest** (destination) object, and **step** is set back to 0. This tells the program that morphing is not happening or it has just finished.

```
// If We're Morphing And We Haven't Gone Through All 200 Steps Increase Our Step (
// Otherwise Set Morphing To False, Make Source=Destination And Set The Step Count
if(morph && step<=steps)step++; else { morph=FALSE; sour=dest; step=0; }</pre>
```

The KillGLWindow() code hasn't changed much. The only real difference is that we free all of the objects from memory before we kill the windows. This prevents memory leaks, and is good practice ;)

```
GLvoid KillGLWindow(GLvoid)
                                       // Properly Kill The Window
{
         objfree(&morph1); // Jump To Code To Release morph1 Allocated Ram
         objfree(&morph2); // Jump To Code To Release morph2 Allocated Ram
         objfree(&morph3); // Jump To Code To Release morph3 Allocated Ram
         objfree(&morph4); // Jump To Code To Release morph4 Allocated Ram objfree(&helper); // Jump To Code To Release helper Allocated Ram
          if (fullscreen) // Are We In Fullscreen Mode?
          {
                   ChangeDisplaySettings(NULL,0);
                                                          // If So Switch Back To The Desktop
                   ShowCursor(TRUE); // Show Mouse Pointer
          }
          if (hRC) // Do We Have A Rendering Context?
          {
                   if (!wglMakeCurrent(NULL,NULL))
                                                         // Are We Able To Release The DC And
                   {
                             MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR",
                   }
```

```
if (!wglDeleteContext(hRC)) // Are We Able To Delete The RC?
        {
                 MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN EF
        hRC=NULL;
                        // Set RC To NULL
}
if (hDC && !ReleaseDC(hWnd,hDC)) // Are We Able To Release The DC
{
        MessageBox(NULL, "Release Device Context Failed.", "SHUTDOWN ERROR", MB_OK
                    // Set DC To NULL
        hDC=NULL;
}
if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?
{
        MessageBox(NULL, "Could Not Release hWnd.", "SHUTDOWN ERROR", MB_OK | MB_ICC
        hWnd=NULL;
                     // Set hWnd To NULL
}
if (!UnregisterClass("OpenGL", hInstance)) // Are We Able To Unregister Class
{
        MessageBox(NULL, "Could Not Unregister Class.", "SHUTDOWN ERROR", MB_OK | ME
        hInstance=NULL; // Set hInstance To NULL
}
```

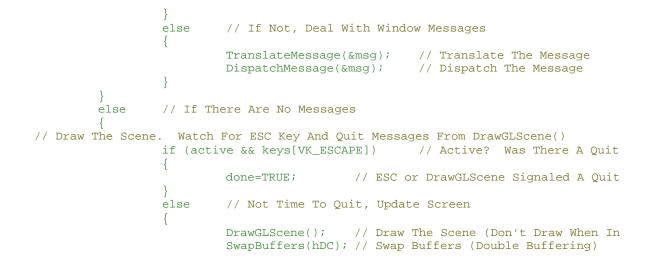
The CreateGLWindow() and WndProc() code hasn't changed. So I'll skip over it.

```
BOOL CreateGLWindow() // Creates The GL Window
LRESULT CALLBACK WndProc() // Handle For This Window
```

}

In WinMain() there are a few changes. First thing to note is the new caption on the title bar :)

```
hInstance, // Instance
hPrevInstance, // Previous Instance
int WINAPI WinMain(
                          HINSTANCE
                          HINSTANCE
                                            lpCmdLine,
                          LPSTR
                                                             // Command Line Parameters
                                            nCmdShow)
                                                              // Window Show State
                          int
{
                         // Windows Message Structure
        MSG
                 msg;
                               // Bool Variable To Exit Loop
        BOOL
                 done=FALSE;
         // Ask The User Which Screen Mode They Prefer
        if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreer
         {
                 fullscreen=FALSE; // Windowed Mode
         }
         // Create Our OpenGL Window
         if (!CreateGLWindow("Piotr Cieslak & NeHe's Morphing Points Tutorial",640,480,16,1
         {
                 return 0;
                                  // Quit If Window Was Not Created
         }
                      // Loop That Runs While done=FALSE
        while(!done)
         {
                  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting
                  {
                           if (msg.message==WM_QUIT) // Have We Received A Quit Message?
                           {
                                                    // If So done=TRUE
                                   done=TRUE;
```



The code below watches for key presses. By now you should understand the code fairly easily. If page up is pressed we increase **zspeed**. This causes the object to spin faster on the z axis in a positive direction.

If page down is pressed we decrease **zspeed**. This causes the object to spin faster on the z axis in a negative direction.

If the down arrow is pressed we increase **xspeed**. This causes the object to spin faster on the x axis in a positive direction.

If the up arrow is pressed we decrease **xspeed**. This causes the object to spin faster on the x axis in a negative direction.

If the right arrow is pressed we increase **yspeed**. This causes the object to spin faster on the y axis in a positive direction.

If the left arrow is pressed we decrease **yspeed**. This causes the object to spin faster on the y axis in a negative direction.

if(keys[VK_PRIOR]) // Is Page Up Being Pressed? zspeed+=0.01f; // Increase zspeed if(keys[VK_NEXT]) // Is Page Down Being Pressed? zspeed-=0.01f; // Decrease zspeed if(keys[VK_DOWN]) // Is Page Up Being Pressed? xspeed+=0.01f; // Increase xspeed if(keys[VK_UP]) // Is Page Up Being Pressed? xspeed==0.01f; // Decrease xspeed if(keys[VK_RIGHT]) // Is Page Up Being Pressed? yspeed+=0.01f; // Increase yspeed if(keys[VK_LEFT]) // Is Page Up Being Pressed? yspeed==0.01f; // Decrease yspeed

The following keys physically move the object. 'Q' moves it into the screen, 'Z' moves it towards the viewer, 'W' moves the object up, 'S' moves it down, 'D' moves it right, and 'A' moves it left.

if (keys['Q']) // Is Q Key Being Pressed? cz-=0.01f; // Move Object Away From Viewer

<pre>if (keys['Z']) cz+=0.01f;</pre>	<pre>// Is Z Key Being Pressed? // Move Object Towards Viewer</pre>
	// Is W Key Being Pressed? // Move Object Up
	// Is S Key Being Pressed? // Move Object Down
<pre>if (keys['D']) cx+=0.01f;</pre>	// Is D Key Being Pressed? // Move Object Right
<pre>if (keys['A']) cx-=0.01f;</pre>	// Is A Key Being Pressed? // Move Object Left

Now we watch to see if keys 1 through 4 are pressed. If 1 is pressed and **key** is not equal to 1 (not the current object already) and **morph** is false (not already in the process of morphing), we set **key** to 1, so that our program knows we just selected object 1. We then set **morph** to TRUE, letting our program know it's time to start morphing, and last we set the destination object (**dest**) to equal object 1 (**morph1**).

Pressing keys 2, 3, and 4 does the same thing. If 2 is pressed we set **dest** to **morph2**, and we set **key** to equal 2. Pressing 3, sets **dest** to **morph3** and **key** to 3.

By setting **key** to the value of the key we just pressed on the keyboard, we prevent the user from trying to morph from a sphere to a sphere or a cone to a cone!

```
if (keys['1'] && (key!=1) && !morph) // Is 1 Pre
{
        key=1; // Sets key To 1 (To Prevent Pressing
        morph=TRUE; // Set morph To True (Starts
                         // Destination Object To Mor
        dest=&morph1;
if (keys['2'] && (key!=2) && !morph)
                                             // Is 2 Pr(
{
         key=2; // Sets key To 2 (To Prevent Pressing
        morph=TRUE; // Set morph To True (Starts
        dest=&morph2;
                          // Destination Object To Mor
if (keys['3'] && (key!=3) && !morph)
                                             // Is 3 Pre
{
        key=3; // Sets key To 3 (To Prevent Pressing
        morph=TRUE; // Set morph To True (Starts
         dest=&morph3;
                          // Destination Object To Mor
                                             // Is 4 Pre
if (keys['4'] && (key!=4) && !morph)
ł
        key=4; // Sets key To 4 (To Prevent Pressing
        morph=TRUE; // Set morph To True (Starts
dest=&morph4; // Destination Object To Mor;
}
```

Finally we watch to see if F1 is pressed if it is we toggle from Fullscreen to Windowed mode or Windowed mode to Fullscreen mode!



I hope you have enjoyed this tutorial. Although it's not an incredibly complex tutorial, you can learn alot from the code! The animation in my dolphin demo is done in a similar way to the morphing in this demo. By playing around with the code you can come up with some really cool effects. Dots turning into words. Faked animation, and more! You may even want to try using solid polygons or lines instead of dots. The effect can be quite impressive!

Piotr's code is new and refreshing. I hope that after reading through this tutorial you have a better understanding on how to store and load object data from a file, and how to manipulate the data to create cool GL effects in your own programs! The .html for this tutorial took 3 days to write. If you notice any mistakes please let me know. Alot of it was written late at night, meaning a few mistakes may have crept in. I want these tutorials to be the best they can be. Feedback is appreciated!

Piotr Cieslak - Code Jeff Molofee (NeHe) - HTML / Modifications

* DOWNLOAD Visual C++ Code For This Lesson.

* DOWNLOAD Mac OS Code For This Lesson. (Conversion by Morgan Aldridge)

Lesson 27

Welcome to another exciting tutorial. The code for this tutorial was written by Banu Cosmin. The tutorial was of course written by myself (NeHe). In this tutorial you will learn how to create EXTREMELY realistic reflections. Nothing fake here! The objects being reflected will not show up underneath the floor or on the other side of a wall. True reflections!

A very important thing to note about this tutorial: Because the Voodoo 1, 2 and some other cards do not support the stencil buffer, this demo will NOT run on those cards. It will ONLY run on cards that support the stencil buffer. If you're not sure if your card supports the stencil buffer, download the code, and try running the demo. Also, this demo requires a fairly decent processor and graphics card. Even on my GeForce I notice there is a little slow down at times. This demo runs best in 32 bit color mode!

As video cards get better, and processors get faster, I can see the stencil buffer becoming more popular. If you have the hardware and you're ready to reflect, read on!

The first part of the code is fairly standard. We include all necessary header files, and set up our Device Context, Rendering Context, etc.

#include	<windows.h></windows.h>	// Header File For Windows
#include	<gl\gl.h></gl\gl.h>	// Header File For The OpenGL32 Library
#include	<gl\glu.h></gl\glu.h>	// Header File For The GLu32 Library
#include	<gl\glaux.h></gl\glaux.h>	// Header File For The Glaux Library
#include	<stdio.h></stdio.h>	// Header File For Standard Input / Output
HDC h	nDC=NULL;	// Private GDI Device Context
HGLRC	hRC=NULL;	// Permanent Rendering Context
HWND	hWnd=NULL;	// Holds Our Window Handle
HINSTANCE	hInstance = NULL;	// Holds The Instance Of The Application

Next we have the standard variables to keep track of key presses (**keys[]**), whether or not the program is active (**active**), and if we should use fullscreen mode or windowed mode (**fullscreen**).

bool	keys[256];	// Array Used For The Keyboard Routine
bool	active=TRUE;	// Window Active Flag Set To TRUE By Default
bool	fullscreen=TRUE;	// Fullscreen Flag Set To Fullscreen Mode By De

Next we set up our lighting variables. **LightAmb[]** will set our ambient light. We will use 70% red, 70% green and 70% blue, creating a light that is 70% bright white. **LightDif[]** will set the diffuse lighting (the amount of light evenly reflected off the surface of our object). In this case we want to reflect full intensity light. Lastly we have **LightPos[]** which will be used to position our light. In this case we want the light 4 units to the right, 4 units up, and 6 units towards the viewer. If we could actually see the light, it would be floating in front of the top right corner of our screen.

// Light Paramet	ers			
static GLfloat	$LightAmb[] = \{0.7f$, 0.7f, 0.7	'f, 1.0f};	// Ambient Light
static GLfloat	$LightDif[] = \{1.0f\}$, 1.0f, 1.0	f, 1.0f};	// Diffuse Light

static GLfloat LightPos[] = {4.0f, 4.0f, 6.0f, 1.0f}; // Light Position

// Declaration For WndP:

We set up a variable called **q** for our quadratic object, **xrot** and **yrot** to keep track of rotation. xrotspeed and yrotspeed control the speed our object rotates at. zoom is used to zoom in and out of the scene (we start at -7 which shows us the entire scene) and height is the height of the ball above the floor.

We then make room for our 3 textures with texture[3], and define WndProc().

GLUquadric	Obj *q;	// Quadratic For Drawing A Sphere
GLfloat GLfloat GLfloat GLfloat GLfloat GLfloat	<pre>xrot = 0.0f; yrot = 0.0f; xrotspeed = 0.0f; yrotspeed = 0.0f; zoom = -7.0f; height = 2.0f;</pre>	<pre>// X Rotation // Y Rotation // X Rotation Speed // Y Rotation Speed // Depth Into The Screen // Height Of Ball From Floor</pre>
GLuint	<pre>texture[3];</pre>	// 3 Textures

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

The ReSizeGLScene() and LoadBMP() code has not changed so I will skip over both sections of code.

GLvoid ReSizeGLScene(GLsizei	width, GLsizei height)	// Resize And Initialize The	
AUX_RGBImageRec *LoadBMP(char	*Filename)	// Loads A Bitmap Image	

The load texture code is pretty standard. You've used it many times before in the previous tutorials. We make room for 3 textures, then we load the three images, and create linear filtered textures from the image data. The bitmap files we use are located in the DATA directory.

```
int LoadGLTextures()
                                                                                                                            // Load Bitmaps And Convert To Textures
{
          memset(TextureImage[3]; // Create Storage Space For The Text
if ((TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-TextureImage[0]-Texture
           int Status=FALSE;
                                                                                                                            // Status Indicator
          if ((TextureImage[0]=LoadBMP("Data/EnvWall.bmp")) && // Load The Floor Texture
                      (TextureImage[1]=LoadBMP("Data/Ball.bmp")) &&
                                                                                                                                                                                        // Load the Light Texture
                      (TextureImage[2]=LoadBMP("Data/EnvRoll.bmp")))
                                                                                                                                                                                          // Load the Wall Texture
         {
                Status=TRUE;
                                                                                                           // Set The Status To TRUE
                 for (int loop=0; loop<3; loop++)
{</pre>
                                                                                                                                            // Create The Texture
                                                                                                                                              // Loop Through 5 Textures
                 {
                        glBindTexture(GL_TEXTURE_2D, texture[loop]);
                        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage[loop]->;
                        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
                        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
                 for (loop=0; loop<3; loop++)</pre>
                                                                                                                                          // Loop Through 5 Textures
                          if (TextureImage[loop])
                                                                                                                                  // If Texture Exists
                          {
                                  if (TextureImage[loop]->data)
                                                                                                                                          // If Texture Image Exists
                                  {
                                          free(TextureImage[loop]->data);
                                                                                                                                              // Free The Texture Image Memory
                                  }
```

```
free(TextureImage[loop]); // Free The Image Structure
}
}
return Status; // Return The Status
}
```

A new command called glClearStencil is introduced in the init code. Passing 0 as a parameter tells OpenGL to disable clearing of the stencil buffer. You should be familiar with the rest of the code by now. We load our textures and enable smooth shading. The clear color is set to an off blue and the clear depth is set to 1.0f. The stencil clear value is set to 0. We enable depth testing, and set the depth test value to less than or equal to. Our perspective correction is set to nicest (very good quality) and 2d texture mapping is enabled.

```
int InitGL(GLvoid)
                                      // All Setup For OpenGL Goes Here
ł
  if (!LoadGLTextures())
                                           // If Loading The Textures Failed
  {
                                    // Return False
     return FALSE;
  }
  glShadeModel(GL_SMOOTH);
                                           // Enable Smooth Shading
  glClearColor(0.2f, 0.5f, 1.0f, 1.0f);
                                              // Background
  glClearDepth(1.0f);
                                        // Depth Buffer Setup
                                       // Clear The Stencil Buffer To 0
  glClearStencil(0);
  glEnable(GL_DEPTH_TEST);
                                          // Enables Depth Testing
  glDepthFunc(GL_LEQUAL);
                                           // The Type Of Depth Testing To Do
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Ca
  glEnable(GL_TEXTURE_2D);
                                          // Enable 2D Texture Mapping
```

Now it's time to set up light 0. The first line below tells OpenGL to use the values stored in **LightAmb** for the Ambient light. If you remember at the beginning of the code, the rgb values of **LightAmb** were all 0.7f, giving us a white light at 70% full intensity. We then set the Diffuse light using the values stored in **LightDif** and position the light using the x,y,z values stored in **LightPos**.

After we have set the light up we can enable it with glEnable(GL_LIGHT0). Even though the light is enabled, you will not see it until we enable lighting with the last line of code.

Note: If we wanted to turn off all lights in a scene we would use glDisable(GL_LIGHTING). If we wanted to disable just one of our lights we would use glDisable(GL_LIGHT{0-7}). This gives us alot of control over the lighting and what lights are on and off. Just remember if GL_LIGHTING is disabled, you will not see lights!

glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmb); // Set The Ambient Lighting For 1
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDif); // Set The Diffuse Lighting For 1
glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Set The Position For Light0
glEnable(GL_LIGHT0); // Enable Light 0
glEnable(GL_LIGHTING); // Enable Lighting

In the first line below, we create a new quadratic object. The second line tells OpenGL to generate smooth normals for our quadratic object, and the third line tells OpenGL to generate texture coordinates for our quadratic. Without the second and third lines of code, our object would use flat shading and we wouldn't be able to texture it.

The fourth and fifth lines tell OpenGL to use the Sphere Mapping algorithm to generate the texture coordinates. This allows us to sphere map the quadratic object.

The code below will draw our object (which is a cool looking environment mapped beach ball).

We set the color to full intensity white and bind to our BALL texture (the ball texture is a series of red, white and blue stripes).

After selecting our texture, we draw a Quadratic Sphere with a radius of 0.35f, 32 slices and 16 stacks (up and down).

```
void DrawObject() // Draw Our Ball
{
  glColor3f(1.0f, 1.0f, 1.0f); // Set Color To White
  glBindTexture(GL_TEXTURE_2D, texture[1]); // Select Texture 2 (1)
  gluSphere(q, 0.35f, 32, 16); // Draw First Sphere
```

After drawing the first sphere, we select a new texture (EnvRoll), set the alpha value to 40% and enable blending based on the source alpha value. glEnable(GL_TEXTURE_GEN_S) and glEnable (GL_TEXTURE_GEN_T) enables sphere mapping.

After doing all that, we redraw the sphere, disable sphere mapping and disable blending.

The final result is a reflection that almost looks like bright points of light mapped to the beach ball. Because we enable sphere mapping, the texture is always facing the viewer, even as the ball spins. We blend so that the new texture doesn't cancel out the old texture (a form of multitexturing).

```
glBindTexture(GL_TEXTURE_2D, texture[2]);
                                                // Select Texture 3 (2)
  glColor4f(1.0f, 1.0f, 1.0f, 0.4f);
                                             // Set Color To White With 40% Alpha
                                     // Enable Blending
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
                                    // Set Blending Mode To Mix Based On SI
  // Enable Sphere Mapping
                                         // Enable Sphere Mapping
  gluSphere(q, 0.35f, 32, 16);
                                          // Draw Another Sphere Using New Texture
                        // Textures Will Mix Creating A MultiTexture Effect (Reflect:
  glDisable(GL_TEXTURE_GEN_S); // Disable Sphere Mapping
  glDisable(GL_TEXTURE_GEN_T);
                                           // Disable Sphere Mapping
  glDisable(GL_BLEND);
                                     // Disable Blending
}
```

The code below draws the floor that our ball hovers over. We select the floor texture (EnvWall), and draw a single texture mapped quad on the z-axis. Pretty simple!

```
void DrawFloor() // Draws The Floor
{
    glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Texture 1 (0)
    glBegin(GL_QUADS); // Begin Drawing A Quad
    glNormal3f(0.0, 1.0, 0.0); // Normal Pointing Up
    glTexCoord2f(0.0f, 1.0f); // Bottom Left Of Texture
    glVertex3f(-2.0, 0.0, 2.0); // Bottom Left Corner Of Floor
```

```
glTexCoord2f(0.0f, 0.0f); // Top Left Of Texture
glVertex3f(-2.0, 0.0,-2.0); // Top Left Corner Of Floor

glTexCoord2f(1.0f, 0.0f); // Top Right Of Texture
glVertex3f( 2.0, 0.0,-2.0); // Top Right Corner Of Floor

glTexCoord2f(1.0f, 1.0f); // Bottom Right Of Texture
glVertex3f( 2.0, 0.0, 2.0); // Bottom Right Corner Of Floor

glEnd(); // Done Drawing The Quad
}
```

Now for the fun stuff. Here's where we combine all the objects and images to create our reflective scene.

We start off by clearing the screen (GL_COLOR_BUFFER_BIT) to our default clear color (off blue). The depth (GL_DEPTH_BUFFER_BIT) and stencil (GL_STENCIL_BUFFER_BIT) buffers are also cleared. Make sure you include the stencil buffer code, it's new and easy to overlook! It's important to note when we clear the stencil buffer, we are filling it with 0's.

After clearing the screen and buffers, we define our clipping plane equation. The plane equation is used for clipping the reflected image.

The equation eqr[]={0.0f, -1.0f, 0.0f, 0.0f} will be used when we draw the reflected image. As you can see, the value for the y-plane is a negative value. Meaning we will only see pixels if they are drawn below the floor or at a negative value on the y-axis. Anything drawn above the floor will not show up when using this equation.

More on clipping later... read on.

```
int DrawGLScene(GLvoid) // Draw Everything
{
    // Clear Screen, Depth Buffer & Stencil Buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
    // Clip Plane Equations
    double eqr[] = {0.0f, -1.0f, 0.0f, 0.0f}; // Plane Equation To Use For The Red
```

So we have cleared the screen, and defined our clipping planes. Now for the fun stuff!

We start off by resetting the modelview matrix. Which of course starts all drawing in the center of the screen. We then translate down 0.6f units (to add a small perspective tilt to the floor) and into the screen based on the value of **zoom**. To better explain why we translate down 0.6f units, I'll explain using a simple example. If you were looking at the side of a piece of paper at exactly eye level, you would barely be able to see it. It would more than likely look like a thin line. If you moved the paper down a little, it would no longer look like a line. You would see more of the paper, because your eyes would be looking down at the page instead of directly at the edge of the paper.

```
glLoadIdentity();
glTranslatef(0.0f, -0.6f, zoom);
```

// Reset The Modelview Matrix
 // Zoom And Raise Camera Above The Floor

Next we set the color mask. Something new to this tutorial! The 4 values for color mask represent red, green, blue and alpha. By default all the values are set to GL_TRUE.

If the red value of glColorMask({red},{green},{blue},{alpha}) was set to GL_TRUE, and all of the other values were 0 (GL_FALSE), the only color that would show up on the screen is red. If the value for red was 0 (GL_FALSE), but the other values were all GL_TRUE, every color except red would be drawn to the screen.

We don't want anything drawn to the screen at the moment, with all of the values set to 0 (GL_FALSE), colors will not be drawn to the screen.

glColorMask(0,0,0,0);

// Set Color Mask

Now even more fun stuff... Setting up the stencil buffer and stencil testing!

We start off by enabling stencil testing. Once stencil testing has been enabled, we are able to modify the stencil buffer.

It's very hard to explain the commands below so please bear with me, and if you have a better explanation, please let me know. In the code below we set up a test. The line glStencilFunc (GL_ALWAYS, 1, 1) tells OpenGL what type of test we want to do on each pixel when an object is drawn to the screen.

GL_ALWAYS just tells OpenGL the test will always pass. The second parameter (1) is a reference value that we will test in the third line of code, and the third parameter is a mask. The mask is a value that is ANDed with the reference value and stored in the stencil buffer when the test is done. A reference value of 1 ANDed with a mask value of 1 is 1. So if the test goes well and we tell OpenGL to, it will place a one in the stencil buffer (reference&mask=1).

Quick note: Stencil testing is a per pixel test done each time an object is drawn to the screen. The reference value ANDed with the mask value is tested against the current stencil value ANDed with the mask value.

The third line of code tests for three different conditions based on the stencil function we decided to use. The first two parameters are GL_KEEP, and the third is GL_REPLACE.

The first parameter tells OpenGL what to do if the test fails. Because the first parameter is GL_KEEP, if the test fails (which it can't because we have the function set to GL_ALWAYS), we would leave the stencil value set at whatever it currently is.

The second parameter tells OpenGL what do do if the stencil test passes, but the depth test fails. In the code below, we eventually disable depth testing so this parameter can be ignored.

The third parameter is the important one. It tells OpenGL what to do if the test passes! In our code we tell OpenGL to replace (GL_REPLACE) the value in the stencil buffer. The value we put into the stencil buffer is our reference value ANDed with our mask value which is 1.

After setting up the type of testing we want to do, we disable depth testing and jump to the code that draws our floor.

In simple english I will try to sum up everything that the code does up until now...

We tell OpenGL not to draw any colors to the screen. This means that when we draw the floor, it wont show up on the screen. BUT... each spot on the screen where the object (our floor) should be if we could see it will be tested based on the type of stencil testing we decide to do. The stencil buffer starts out full of 0's (empty). We want to set the stencil value to 1 wherever our object would have been drawn if we could see it. So we tell OpenGL we don't care about testing. If a pixel should have been drawn to the screen, we want that spot marked with a 1. GL_ALWAYS does exactly that. Our reference and mask values of 1 make sure that the value placed into the stencil buffer is indeed going to be 1! As we invisibly draw, our stencil operation checks each pixel location, and replaces the 0 with a 1.

DrawFloor();

// Draw The Floor (Draws To The Stencil Buffer)
// We Only Want To Mark It In The Stencil Buffer

So now we have an invisible stencil mask of the floor. As long as stencil testing is enabled, the only places pixels will show up are places where the stencil buffer has a value of 1. All of the pixels on the screen where the invisible floor was drawn will have a stencil value of 1. Meaning as long as stencil testing is enabled, the only pixels that we will see are the pixels that we draw in the same spot our invisible floor was defined in the stencil buffer. The trick behind creating a real looking reflection that reflects in the floor and nowhere else!

So now that we know the ball reflection will only be drawn where the floor should be, it's time to draw the reflection! We enable depth testing, and set the color mask back to all ones (meaning all the colors will be drawn to the screen).

Instead of using GL_ALWAYS for our stencil function we are going to use GL_EQUAL. We'll leave the reference and mask values at 1. For the stencil operation we will set all the parameters to GL_KEEP. In english, any object we draw this time around will actually appear on the screen (because the color mask is set to true for each color). As long as stencil testing is enabled pixels will ONLY be drawn if the stencil buffer has a value of 1 (reference value ANDed with the mask, which is 1 EQUALS (GL_EQUAL) the stencil buffer value ANDed with the mask, which is also 1). If the stencil value is not 1 where the current pixel is being drawn it will not show up! GL_KEEP just tells OpenGL not to modify any values in the stencil buffer if the test passes OR fails!

Now we enable the mirrored clipping plane. This plane is defined by **eqr**, and only allows object to be drawn from the center of the screen (where the floor is) down to the bottom of the screen (any negative value on the y-axis). That way the reflected ball that we draw can't come up through the center of the floor. That would look pretty bad if it did. If you don't understand what I mean, remove the first line below from the source code, and move the real ball (non reflected) through the floor. If clipping is not enabled, you will see the reflected ball pop out of the floor as the real ball goes into the floor.

After we enable clipping plane0 (usually you can have from 0-5 clipping planes), we define the plane by telling it to use the parameters stored in **eqr**.

We push the matrix (which basically saves the position of everything on the screen) and use glScalef (1.0f, -1.0f, 1.0f) to flip the object upside down (creating a real looking reflection). Setting the y value of glScalef({x},{y},{z}) to a negative value forces OpenGL to render opposite on the y-axis. It's almost like flipping the entire screen upside down. When position an object at a positive value on the y-axis, it will appear at the bottom of the screen instead of at the top. When you rotate an object towards yourself, it will rotate away from you. Everything will be mirrored on the y-axis until you pop the matrix or set the y value back to 1.0f instead of -1.0f using glScalef({x},{y},{z}).

glEnable(GL_CLIP_PLANE0);	// Enable Clip Plane For Removing Artifacts
// (When T	he Object Crosses The Floor)
glClipPlane(GL_CLIP_PLANE0, eqr);	<pre>// Equation For Reflected Objects</pre>
glPushMatrix();	// Push The Matrix Onto The Stack
glScalef(1.0f, -1.0f, 1.0f);	// Mirror Y Axis

The first line below positions our light to the location specified by **LightPos**. The light should shine on the bottom right of the reflected ball creating a very real looking light source. The position of the light is also mirrored. On the real ball (ball above the floor) the light is positioned at the top right of your screen, and shines on the top right of the real ball. When drawing the reflected ball, the light is

positioned at the bottom right of your screen.

We then move up or down on the y-axis to the value specified by **height**. Translations are mirrored, so if the value of height is 5.0f, the position we translate to will be mirrored (-5.0f). Positioning the reflected image under the floor, instead of above the floor!

After position our reflected ball, we rotate the ball on both the x axis and y axis, based on the values of **xrot** and **yrot**. Keep in mind that any rotations on the x axis will also be mirrored. So if the real ball (ball above the floor) is rolling towards you on the x-axis, it will be rolling away from you in the reflection.

After positioning the reflected ball and doing our rotations we draw the ball by calling DrawObject(), and pop the matrix (restoring things to how they were before we drew the ball). Popping the matrix all cancels mirroring on the y-axis.

We then disable our clipping plane (plane0) so that we are not stuck drawing only to the bottom half of the screen, and last, we disable stencil testing so that we can draw to other spots on the screen other than where the floor should be.

Note that we draw the reflected ball before we draw the floor. I'll explain why later on.

```
glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Set Up Light0
glTranslatef(0.0f, height, 0.0f); // Position The Object
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Rotate Local Coordinate System On X
glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Rotate Local Coordinate System On Y
DrawObject(); // Draw The Sphere (Reflection)
glPopMatrix(); // Pop The Matrix Off The Stack
glDisable(GL_CLIP_PLANE0); // Disable Clip Plane For Drawing The Floor
glDisable(GL_STENCIL_TEST); // We Don't Need The Stencil Buffer Any Moj
```

We start off this section of code by positioning our light. The y-axis is no longer being mirrored so drawing the light this time around will position it at the top of the screen instead of the bottom right of the screen.

We enable blending, disable lighting, and set the alpha value to 80% using the command glColor4f (1.0f,1.0f,0.8f). The blending mode is set up using glBlendFunc(), and the semi transparent floor is drawn over top of the reflected ball.

If we drew the floor first and then the reflected ball, the effect wouldn't look very good. By drawing the ball and then the floor, you can see a small amount of coloring from the floor mixed into the coloring of the ball. If I was looking into a BLUE mirror, I would expect the reflection to look a little blue. By rendering the ball first, the reflected image looks like it's tinted the color of the floor.

```
glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Set Up Light0 Position
glEnable(GL_BLEND); // Enable Blending (Otherwise The Reflected Obj
glDisable(GL_LIGHTING); // Since We Use Blending, We Disable Lightir
glColor4f(1.0f, 1.0f, 1.0f, 0.8f); // Set Color To White With 80% Alpha
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Blending Based On Source i
DrawFloor(); // Draw The Floor To The Screen
```

Now we draw the 'real' ball (the one that floats above the floor). We disabled lighting when we drew the floor, but now it's time to draw another ball so we will turn lighting back on.

We don't need blending anymore so we disable blending. If we didn't disable blending, the colors from the floor would mix with the colors of our 'real' ball when it was floating over top of the floor. We don't want the 'real' ball to look like the reflection so we disable blending.

We are not going to clip the actual ball. If the real ball goes through the floor, we should see it come out the bottom. If we were using clipping the ball wouldn't show up after it went through the floor. If you didn't want to see the ball come through the floor, you would set up a clipping equation that set

the Y value to +1.0f, then when the ball went through the floor, you wouldn't see it (you would only see the ball when it was drawn on at a positive value on the y-axis. For this demo, there's no reason we shouldn't see it come through the floor.

We then translate up or down on the y-axis to the position specified by **height**. Only this time the y-axis is not mirrored, so the ball travels the opposite direction that the reflected image travels. If we move the 'real' ball down the reflected ball will move up. If we move the 'real' ball up, the reflected ball will move down.

We rotate the 'real' ball, and again, because the y-axis is not mirrored, the ball will spin the opposite direction of the reflected ball. If the reflected ball is rolling towards you the 'real' ball will be rolling away from you. This creates the illusion of a real reflection.

After positioning and rotating the ball, we draw the 'real' ball by calling DrawObject().

```
glEnable(GL_LIGHTING); // Enable Lighting
glDisable(GL_BLEND); // Disable Blending
glTranslatef(0.0f, height, 0.0f); // Position The Ball At Proper Height
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Rotate On The X Axis
glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Rotate On The Y Axis
DrawObject(); // Draw The Ball
```

The following code rotates the ball on the x and y axis. By increasing **xrot** by **xrotspeed** we rotate the ball on the x-axis. By increasing **yrot** by **yrotspeed** we spin the ball on the y-axis. If **xrotspeed** is a very high value in the positive or negative direction the ball will spin quicker than if **xrotspeed** was a low value, closer to 0.0f. Same goes for **yrotspeed**. The higher the value, the faster the ball spins on the y-axis.

Before we return TRUE, we do a glFlush(). This tells OpenGL to render everything left in the GL pipeline before continuing, and can help prevent flickering on slower video cards.

```
xrot += xrotspeed; // Update X Rotation Angle By xrotspeed
yrot += yrotspeed; // Update Y Rotation Angle By yrotspeed
glFlush(); // Flush The GL Pipeline
return TRUE; // Everything Went OK
```

}

The following code will watch for key presses. The first 4 lines check to see if you are pressing one of the 4 arrow keys. If you are, the ball is spun right, left, down or up.

The next 2 lines check to see if you are pressing the 'A' or 'Z' keys. Pressing 'A' will zoom you in closer to the ball and pressing 'Z' will zoom you away from the ball.

Pressing 'PAGE UP' will increase the value of **height** moving the ball up, and pressing 'PAGE DOWN' will decrease the value of **height** moving the ball down (closer to the floor).

```
void ProcessKeyboard()
                                                        // Process Keyboard Results
ł
   if (keys[VK_RIGHT]) yrotspeed += 0.08f;
if (keys[VK_LEFT]) yrotspeed -= 0.08f;
if (keys[VK_DOWN]) xrotspeed += 0.08f;
                                                                   // Right Arrow Pressed (Increase y)
                                                               // Left Arrow Pressed (Decrease yrot
// Down Arrow Pressed (Ingrease yrot
                                                                  // Down Arrow Pressed (Increase xrot
   if (keys[VK_UP]) xrotspeed -= 0.08f;
                                                              // Up Arrow Pressed (Decrease xrotspe
                                                           // 'A' Key Pressed ... Zoom In
   if (keys['A']) zoom +=0.05f;
if (keys['Z']) zoom -=0.05f;
                                                            // 'Z' Key Pressed ... Zoom Out
   if (keys[VK_PRIOR]) height +=0.03f;
                                                                  // Page Up Key Pressed Move Ball Up
   if (keys[VK_NEXT]) height -=0.03f;
                                                                 // Page Down Key Pressed Move Ball Do
```

}

The KillGLWindow() code hasn't changed, so I'll skip over it.

GLvoid KillGLWindow(GLvoid)

// Properly Kill The Window

You can skim through the following code. Even though only one line of code has changed in CreateGLWindow(), I have included all of the code so it's easier to follow through the tutorial.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
ł
   GLuint
               PixelFormat;
                                                   // Holds The Results After Searching For A Mat
   WNDCLASS wc;
                                           // Windows Class Structure
   DWORD dwExStyle;
DWORD dwStyle;
                                               // Window Extended Style
                                             // Window Style
   fullscreen=fullscreenflag;
                                                     // Set The Global Fullscreen Flag
                  = GetModuleHandle(NULL); // Grab An Instance For Our Window
   hInstance= GetModuleHandle(NULL);// Grab An Instance For Our Windowwc.style= CS_HREDRAW | CS_VREDRAW | CS_OWNDC;// Redraw On Size, And Own DC H
   wc.lpfnWndProc = (WNDPROC) WndProc;
wc.cbClsExtra = 0; // No Ext
                                                              // WndProc Handles Messages
   wc.cbClsExtra= 0;// No Extra Window Datawc.cbWndExtra= 0;// No Extra Window Datawc.hInstance= hInstance;// Set The Instance
   wc.hInstance = hInstance; // Set The Instance
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Load The Default Icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Load The Arrow Pointer
                                    // No Background Required For GL
   wc.hbrBackground = NULL;
   wc.lpszMenuName = NULL;
                                                       // We Don't Want A Menu
   wc.lpszClassName = "OpenGL";
                                                      // Set The Class Name
   if (!RegisterClass(&wc))
                                                   // Attempt To Register The Window Class
   {
      MessageBox(NULL, "Failed To Register The Window Class.", "ERROR", MB_OK MB_ICONEXCLAMAT:
      return FALSE;
                                           // Return FALSE
   }
   if (fullscreen)
                                               // Attempt Fullscreen Mode?
   {
      DEVMODE dmScreenSettings;
                                                   // Device Mode
      memset(&dmScreenSettings,0,sizeof(dmScreenSettings));
                                                                         // Makes Sure Memory's Cle
      dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode Structu
      dmScreenSettings.dmPelsWidth = width; // Selected Screen Width
dmScreenSettings.dmPelsHeight = height; // Selected Screen Height
dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pixel
      dmScreenSettings.dmFields=DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;
       // Try To Set Selected Mode And Get Results. NOTE: CDS_FULLSCREEN Gets Rid Of Start
      if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
       {
          // If The Mode Fails, Offer Two Options. Quit Or Use Windowed Mode
          if (MessageBox(NULL, "The Requested Fullscreen Mode Is Not Supported By\nYour Vide
          {
                                              // Windowed Mode Selected. Fullscreen = FALSE
             fullscreen=FALSE;
          }
          else
          {
             // Pop Up A Message Box Letting User Know The Program Is Closing
             MessageBox(NULL, "Program Will Now Close.", "ERROR", MB_OK MB_ICONSTOP);
             return FALSE;
                                 // Return FALSE
          }
      }
   }
```

```
if (fullscreen)
                                                   // Are We Still In Fullscreen Mode?
{
   dwExStyle=WS_EX_APPWINDOW;
                                                          // Window Extended Style
   dwStyle=WS_POPUP | WS_CLIPSIBLINGS | WS_CLIPCHILDREN; // Windows Style
   ShowCursor(FALSE);
                                                   // Hide Mouse Pointer
}
else
{
   dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // Window Extended Style
   dwStyle=WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN; // Windows Style
}
// Create The Window
if (!(hWnd=CreateWindowEx( dwExStyle,
                                                                  // Extended Style For The Window
               CreateWindowEx(dwExStyle,// Extended Style"OpenGL",// Class Nametitle,// Window TitledwStyle,// Window Style0, 0,// Window Positionwidth, height,// Selected Width And HeightNULL,// No Parent WindowNULL,// No MenuhInstance,// InstanceNULL)))// Dont Pass Anything To WM_CREATE
{
   KillGLWindow();
                                                  // Reset The Display
   MessageBox(NULL,"Window Creation Error.","ERROR",MB_OK MB_ICONEXCLAMATION);
   return FALSE;
                                                // Return FALSE
}
static PIXELFORMATDESCRIPTOR pfd=
                                                                 // pfd Tells Windows How We Want Thing
{
   sizeof(PIXELFORMATDESCRIPTOR),
                                                                // Size Of This Pixel Format Descriptor
  Size:
1,
PFD_DRAW_TO_WINDOW |
PFD_SUPPORT_OPENGL |
PFD_DOUBLEBUFFER, // Must Supp
PFD_TYPE_RGBA, // Request An RGBA F
bits, // Select Our Color Depth
^ 0, 0, 0, 0, 0, 0, // Color Bits Ignore
// No Alpha Buffer
// Shift Bit Ignored
Proumulation Buffer
Tion Bits I
                                                      // Format Must Support Window
                                                       // Format Must Support OpenGL
                                                  // Must Support Double Buffering
                                                  // Request An RGBA Format
                                               // Color Bits Ignored
                                               // Accumulation Bits Ignored
    0, 0, 0, 0,
                                      // 16Bit Z-Buffer (Depth Buffer)
   16,
```

The only change in this section of code is the line below. It is ***VERY IMPORTANT*** you change the value from 0 to 1 or some other non zero value. In all of the previous tutorials the value of the line below was 0. In order to use Stencil Buffering this value HAS to be greater than or equal to 1. This value is the number of bits you want to use for the stencil buffer.

```
1,
                          // Use Stencil Buffer ( * Important * )
                          // No Auxiliary Buffer
  0.
  PFD_MAIN_PLANE,
                                    // Main Drawing Layer
                           // Reserved
  Ο,
                             // Layer Masks Ignored
  0, 0, 0
};
if (!(hDC=GetDC(hWnd)))
                                          // Did We Get A Device Context?
{
                                    // Reset The Display
  KillGLWindow();
  MessageBox(NULL, "Can't Create A GL Device Context.", "ERROR", MB_OK MB_ICONEXCLAMATION
  return FALSE;
                                // Return FALSE
}
if (!(PixelFormat=ChoosePixelFormat(hDC,&pfd))) // Did Windows Find A Matchin
{
```

```
// Reset The Display
   KillGLWindow();
   MessageBox(NULL, "Can't Find A Suitable PixelFormat.", "ERROR", MB_OK MB_ICONEXCLAMATION
                                     // Return FALSE
   return FALSE;
}
if(!SetPixelFormat(hDC,PixelFormat,&pfd))
                                                     // Are We Able To Set The Pixel For
{
                                       // Reset The Display
   KillGLWindow();
   MessageBox(NULL,"Can't Set The PixelFormat.","ERROR",MB_OK MB_ICONEXCLAMATION);
                           // Return FALSE
   return FALSE;
}
if (!(hRC=wqlCreateContext(hDC)))
                                                 // Are We Able To Get A Rendering Conte:
{
                                      // Reset The Display
   KillGLWindow();
  MessageBox(NULL, "Can't Create A GL Rendering Context.", "ERROR", MB_OK MB_ICONEXCLAMATI
   return FALSE;
                                    // Return FALSE
}
if(!wglMakeCurrent(hDC,hRC))
                                               // Try To Activate The Rendering Context
{
                                 // Reset The Display
   KillGLWindow();
   MessageBox(NULL, "Can't Activate The GL Rendering Context.", "ERROR", MB_OK MB_ICONEXCLi
  return FALSE;
                                    // Return FALSE
}
ShowWindow(hWnd,SW_SHOW);
                                            // Show The Window
SetForegroundWindow(hWnd);
SetFocus(hWnd);
ReSizeGLScene(width, height);
                                            // Slightly Higher Priority
                                      // Sets Keyboard Focus To The Window
                                                // Set Up Our Perspective GL Screen
if (!InitGL())
                                      // Initialize Our Newly Created GL Window
{
  KillGLWindow();
                                       // Reset The Display
  MessageBox(NULL, "Initialization Failed.", "ERROR", MB_OK MB_ICONEXCLAMATION);
                                    // Return FALSE
  return FALSE;
}
return TRUE;
                                   // Success
```

WndProc() has not changed, so we will skip over it.

}

LRESULT CALLBACH	(WndProc(HWND	hWnd,	// Handle For This Window
UIN	ſ uMsg,		11	Message For This Window
WPAH	RAM wParam	,		// Additional Message Information
LPA	RAM lParam)		// Additional Message Information

Nothing new here. Typical start to WinMain().

```
fullscreen=FALSE;
}
```

}

// Windowed Mode

The only real big change in this section of the code is the new window title to let everyone know the tutorial is about reflections using the stencil buffer. Also notice that we pass the **resx**, **resy** and **resbpp** variables to our window creation procedure instead of the usual 640, 480 and 16.

```
// Create Our OpenGL Window
if (!CreateGLWindow("Banu Octavian & NeHe's Stencil & Reflection Tutorial", resx, resy,
{
  return 0;
                               // Ouit If Window Was Not Created
}
while(!done)
                                  // Loop That Runs While done=FALSE
{
   if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
                                                 // Is There A Message Waiting?
   {
      if (msg.message==WM_QUIT)
                                        // Have We Received A Quit Message?
     {
        done=TRUE;
                       // If So done=TRUE
     }
                        // If Not, Deal With Window Messages
     else
     {
        TranslateMessage(&msg);
                                         // Translate The Message
        DispatchMessage(&msg);
                                        // Dispatch The Message
     }
   }
                            // If There Are No Messages
  else
   {
      // Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
     if (active) // Program Active?
      {
        if (keys[VK_ESCAPE])
                                       // Was Escape Pressed?
        {
                              // ESC Signalled A Quit
           done=TRUE;
        }
        else
                            // Not Time To Quit, Update Screen
        {
           DrawGLScene();
SwapBuffers(hDC);
                                   // Draw The Scene
                                   // Swap Buffers (Double Buffering)
```

Instead of checking for key presses in WinMain(), we jump to our keyboard handling routine called ProcessKeyboard(). Notice the ProcessKeyboard() routine is only called if the program is active!

```
ProcessKeyboard(); // Processed Keyboard Presses
}
}
// Shutdown
KillGLWindow(); // Kill The Window
return (msg.wParam); // Exit The Program
```

I really hope you've enjoyed this tutorial. I know it could use a little more work. It was one of the more difficult tutorials that I have written. It's easy for me to understand what everything is doing, and what commands I need to use to create cool effects, but when you sit down and actually try to explain things keeping in mind that most people have never even heard of the stencil buffer, it's tough! If you notice anything that could be made clearer or if you find any mistakes in the tutorial

please let me know. As always, I want this tutorial to be the best it can possibly be, your feedback is greatly appreciated.

Banu Cosmin (Choko) - Code Jeff Molofee (NeHe) - HTML / Modifications

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Visual C++ / OpenIL Code For This Lesson. (Conversion by Denton Woods)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Marc Aarts)
- * DOWNLOAD Irix / GLUT Code For This Lesson. (Conversion by Rob Fletcher)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Morgan Aldridge)

Lesson 28

Welcome to a fairly complex tutorial on shadow casting. The effect this demo creates is literally incredible. Shadows that stretch, bend and wrap around other objects and across walls. Everything in the scene can be moved around in 3D space using keys on the keyboard.

This tutorial takes a fairly different approach - It assumes you have a lot of OpenGL knowledge. You should already understand the stencil buffer, and basic OpenGL setup. If you need to brush up, go back and read the earlier tutorials. Functions such as **CreateGLWindow** and **WinMain** will NOT be explained in this tutorial. Additionally, some fundamental 3D math is assumed, so keep a good textbook handy! (I used my 1st year maths lecture notes from University - I knew they'd come in handy later on! :)

First we have the definition of **INFINITY**, which represents how far to extend the shadow volume polygons (this will be explained more later on). If you are using a larger or smaller coordinate system, adjust this value accordingly.

// Definition Of "INFINITY" For Calculating The Extension Vector For The Shadow Volume #define INFINITY 100

Next is the definition of the object structures.

The **Point3f** structure holds a coordinate in 3D space. This can be used for vertices or vectors.

```
// Structure Describing A Vertex In An Object
struct Point3f
{
    GLfloat x, y, z;
};
```

The **Plane** structure holds the 4 values that form the equation of a plane. These planes will represent the faces of the object.

```
// Structure Describing A Plane, In The Format: ax + by + cz + d = 0
struct Plane
{
    GLfloat a, b, c, d;
};
```

The Face structure contains all the information necessary about a triangle to cast a shadow.

- The indices specified are from the object's array of vertices.
- The vertex normals are used to calculate the orientation of the face in 3D space, so you can determine which are facing the light source when casting the shadows.
- The plane equation describes the plane that this triangle lies in, in 3D space.
- The neighbour indices are indices into the array of faces in the object. This allows you to

specify which face joins this face at each edge of the triangle.

The visible parameter is used to specify whether the face is "visible" to the light source which
is casting the shadows.

```
// Structure Describing An Object's Face
struct Face
{
    int vertexIndices[3]; // Index Of Each Vertex Within An Object That Makes Up The
    Point3f normals[3]; // Normals To Each Vertex
    Plane planeEquation; // Equation Of A Plane That Contains This Triangle
    int neighbourIndices[3]; // Index Of Each Face That Neighbours This One Within The
    bool visible; // Is The Face Visible By The Light?
};
```

Finally, the **ShadowedObject** structure contains all the vertices and faces in the object. The memory for each of the arrays is dynamically created when it is loaded.

```
struct ShadowedObject
{
    int nVertices;
    Point3f *pVertices; // Will Be Dynamically Allocated
    int nFaces;
    Face *pFaces; // Will Be Dynamically Allocated
};
```

The **readObject** function is fairly self explanatory. It will fill in the given object structure with the values read from the file, allocating memory for the vertices and faces. It also initializes the neighbours to -1, which means there isn't one (yet). They will be calculated later.

```
bool readObject( const char *filename, ShadowedObject& object )
   FILE *pInputFile;
   int i;
   pInputFile = fopen( filename, "r" );
   if ( pInputFile == NULL )
   {
      cerr << "Unable to open the object file: " << filename << endl;
      return false;
   }
   // Read Vertices
   fscanf( pInputFile, "%d", &object.nVertices );
   object.pVertices = new Point3f[object.nVertices];
   for ( i = 0; i < object.nVertices; i++ )</pre>
      fscanf( pInputFile, "%f", &object.pVertices[i].x );
      fscanf( pInputFile, "%f", &object.pVertices[i].y );
fscanf( pInputFile, "%f", &object.pVertices[i].z );
   }
   // Read Faces
   fscanf( pInputFile, "%d", &object.nFaces );
   object.pFaces = new Face[object.nFaces];
   for ( i = 0; i < object.nFaces; i++ )
      int j;
      Face *pFace = &object.pFaces[i];
```

```
for ( j = 0; j < 3; j++ )
    pFace->neighbourIndices[j] = -1; // No Neigbours Set Up Yet
for ( j = 0; j < 3; j++ )
{
    fscanf( pInputFile, "%d", &pFace->vertexIndices[j] );
    pFace->vertexIndices[j]--; // Files Specify Them With A 1 Array Base, But We
}
for ( j = 0; j < 3; j++ )
{
    fscanf( pInputFile, "%f", &pFace->normals[j].x );
    fscanf( pInputFile, "%f", &pFace->normals[j].y );
    fscanf( pInputFile, "%f", &pFace->normals[j].z );
}
return true;
```

Likewise, **killObject** is self-explanatory - just delete all those dynamically allocated arrays inside the object when you are done with them. Note that a line was added to KillGLWindow to call this function for the object in question.

```
void killObject( ShadowedObject& object )
{
    delete[] object.pFaces;
    object.pFaces = NULL;
    object.nFaces = 0;
    delete[] object.pVertices;
    object.pVertices = NULL;
    object.nVertices = 0;
}
```

}

Now, with **setConnectivity** it starts to get interesting. This function is used to find out what neighbours there are to each face of the object given. Here's some pseudo code:

```
for each face (A) in the object
  for each edge in A
    if we don't know this edges neighbour yet
    for each face (B) in the object (except A)
        for each edge in B
            if A's edge is the same as B's edge, then they are neighbouring each other (
                set the neighbour property for each face A and B, then move onto next edge
```

The last two lines are accomplished with the following code. By finding the two vertices that mark the ends of an edge and comparing them, you can discover if it is the same edge. The part (edgeA+1)% 3 gets a vertex next to the one you are considering. Then you check if the vertices match (the order may be different, hence the second case of the if statement).

```
int vertA1 = pFaceA->vertexIndices[edgeA];
int vertA2 = pFaceA->vertexIndices[( edgeA+1 )%3];
int vertB1 = pFaceB->vertexIndices[edgeB];
int vertB2 = pFaceB->vertexIndices[( edgeB+1 )%3];
// Check If They Are Neighbours - IE, The Edges Are The Same
if (( vertA1 == vertB1 && vertA2 == vertB2 ) || ( vertA1 == vertB2 && vertA2 == vertB1
{
```

```
pFaceA->neighbourIndices[edgeA] = faceB;
pFaceB->neighbourIndices[edgeB] = faceA;
edgeFound = true;
break;
}
```

Luckily, another easy function while you take a breath. drawObject renders each face one by one.

```
// Draw An Object - Simply Draw Each Triangular Face.
void drawObject( const ShadowedObject& object )
{
  glBegin( GL_TRIANGLES );
  for ( int i = 0; i < object.nFaces; i++ )
  {
    const Face& face = object.pFaces[i];
    for ( int j = 0; j < 3; j++ )
    {
      const Point3f& vertex = object.pVertices[face.vertexIndices[j]];
      glNormal3f( face.normals[j].x, face.normals[j].y, face.normals[j].z );
      glVertex3f( vertex.x, vertex.y, vertex.z );
      }
    }
    glEnd();
}
```

Calculating the equation of a plane looks ugly, but it is just a simple mathematical formula that you grab from a textbook when you need it.

Have you caught your breath yet? Good, because you are about to learn how to cast a shadow! The **castShadow** function does all of the GL specifics, and passes it on to **doShadowPass** to render the shadow in two passes.

First up, we determine which surfaces are facing the light. We do this by seeing which side of the plane the light is on. This is done by substituting the light's position into the equation for the plane. If this is larger than 0, then it is in the same direction as the normal to the plane and visible by the light. If not, then it is not visible by the light. (Again, refer to a good Math textbook for a better explanation of geometry in 3D).

```
void castShadow( ShadowedObject& object, GLfloat *lightPosition )
{
    // Determine Which Faces Are Visible By The Light.
```

```
for ( int i = 0; i < object.nFaces; i++ )
{
    const Plane& plane = object.pFaces[i].planeEquation;
    GLfloat side = plane.a*lightPosition[0]+
        plane.b*lightPosition[1]+
        plane.c*lightPosition[2]+
        plane.d;
    if ( side > 0 )
        object.pFaces[i].visible = true;
    else
        object.pFaces[i].visible = false;
}
```

The next section sets up the necessary OpenGL states for rendering the shadows.

First, we push all the attributes onto the stack that will be modified. This makes changing them back a lot easier.

Lighting is disabled because we will not be rendering to the color (output) buffer, just the stencil buffer. For the same reason, the color mask turns off all color components (so drawing a polygon won't get through to the output buffer).

Although depth testing is still used, we don't want the shadows to appear as solid objects in the depth buffer, so the depth mask prevents this from happening.

The stencil buffer is turned on as that is what is going to be used to draw the shadows into.

Ok, now the shadows are actually rendered. We'll come back to that in a moment when we look at the **doShadowPass** function. They are rendered in two passes as you can see, one incrementing the stencil buffer with the front faces (casting the shadow), the second decrementing the stencil buffer with the backfaces ("turning off" the shadow between the object and any other surfaces).

```
// First Pass. Increase Stencil Value In The Shadow
glFrontFace( GL_CCW );
glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );
doShadowPass( object, lightPosition );
// Second Pass. Decrease Stencil Value In The Shadow
glFrontFace( GL_CW );
glStencilOp( GL_KEEP, GL_KEEP, GL_DECR );
doShadowPass( object, lightPosition );
```

To understand how the second pass works, my best advise is to comment it out and run the tutorial again. To save you the trouble, I have done it here:

}

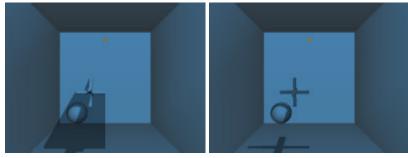


Figure 1: First Pass

Figure 2: Second Pass

The final section of this function draws one blended rectangle over the whole screen, to cast a shadow. The darker you make this rectangle, the darker the shadows will be. So to change the properties of the shadow, change the **glColor4f** statement. Higher alpha will make it more black. Or you can make it red, green, purple, ...!

```
glFrontFace( GL_CCW );
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE ); // Enable Rendering To Colour Buff(
// Draw A Shadowing Rectangle Covering The Entire Screen
glColor4f( 0.0f, 0.0f, 0.0f, 0.4f );
glEnable( GL_BLEND );
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
glStencilFunc( GL_NOTEQUAL, 0, 0xFFFFFFFFL );
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
glPushMatrix();
glLoadIdentity();
glBegin( GL_TRIANGLE_STRIP );
  glVertex3f(-0.1f, 0.1f,-0.10f);
   glVertex3f(-0.1f,-0.1f,-0.10f);
  glVertex3f( 0.1f, 0.1f, -0.10f);
  glVertex3f( 0.1f, -0.1f, -0.10f);
glEnd();
glPopMatrix();
glPopAttrib();
```

Ok, the next part draws the shadowed quads. How does that work? What happens is that you go through every face, and if it is visible, then you check all of its edges. If at the edge, there is no neighbouring face, or the neighbouring face is not visible, the edge casts a shadow. If you think about the two cases clearly, then you'll see this is true. By drawing a quadrilateral (as two triangles) comprising of the points of the edge, and the edge projected backwards through the scene you get the shadow cast by it.

The brute force approach used here just draws to "infinity", and the shadow polygon is clipped against all the polygons it encounters. This causes piercing, which will stress the video hardware. For a high-performance modification to this algorithm, you should clip the polygon to the objects behind it. This is much trickier and has problems of its own, but if that's what you want to do, you should refer to this Gamasutra article.

The code to do all of that is not as tricky as it sounds. To start with, here is a snippet that loops through the objects. By the end of it, we have an edge, *j*, and its neighbouring face, specified by *neighbourIndex*.

```
void doShadowPass( ShadowedObject& object, GLfloat *lightPosition )
{
   for ( int i = 0; i < object.nFaces; i++ )
   {
      const Face& face = object.pFaces[i];
   }
}</pre>
```

```
if ( face.visible )
{
    // Go Through Each Edge
    for ( int j = 0; j < 3; j++ )
    {
        int neighbourIndex = face.neighbourIndices[j];
    }
}</pre>
```

Next, check if there is a visible neighbouring face to this object. If not, then this edge casts a shadow.

```
// If There Is No Neighbour, Or Its Neighbouring Face Is Not Visible, Then This
if ( neighbourIndex == -1 || object.pFaces[neighbourIndex].visible == false )
{
```

The next segment of code will retrieve the two vertices from the current edge, v1 and v2. Then, it calculates v3 and v4, which are projected along the vector between the light source and the first edge. They are scaled to **INFINITY**, which was set to a very large value.

// Get The Points On The Edge const Point3f& v1 = object.pVertices[face.vertexIndices[j]]; const Point3f& v2 = object.pVertices[face.vertexIndices[(j+1)%3]]; // Calculate The Two Vertices In Distance Point3f v3, v4; v3.x = (v1.x-lightPosition[0])*INFINITY; v3.y = (v1.y-lightPosition[1])*INFINITY; v3.z = (v1.z-lightPosition[2])*INFINITY; v4.x = (v2.x-lightPosition[0])*INFINITY; v4.y = (v2.y-lightPosition[1])*INFINITY; v4.z = (v2.z-lightPosition[2])*INFINITY;

I think you'll understand the next section, it justs draws the quadrilateral defined by those four points:

```
// Draw The Quadrilateral (As A Triangle Strip)
glBegin( GL_TRIANGLE_STRIP );
    glVertex3f( v1.x, v1.y, v1.z );
    glVertex3f( v1.x+v3.x, v1.y+v3.y, v1.z+v3.z );
    glVertex3f( v2.x, v2.y, v2.z );
    glVertex3f( v2.x+v4.x, v2.y+v4.y, v2.z+v4.z );
    glEnd();
    }
}
```

With that, the shadow casting section is completed. But we are not finished yet! What about **drawGLScene**? Lets start with the simple bits: clearing the buffers, positioning the light source, and drawing a sphere:

}

{

```
GLmatrix16f Minv;
GLvector4f wlp, lp;
// Clear Color Buffer, Depth Buffer, Stencil Buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glLoadIdentity(); // Reset Modelview Matrix
glLightfv(GL_LIGHT1, GL_POSITION, LightPos); // Position Light1
glTranslatef(0.0f, 0.0f, -20.0f); // Zoom Into Screen 20 Units
glTranslatef(SpherePos[0], SpherePos[1], SpherePos[2]); // Position The Sphere
gluSphere(q, 1.5f, 32, 16); // Draw A Sphere
```

Next, we have to calculate the light's position relative to the local coordinate system of the object. The comments explain each step in detail. *Minv* stores the object's transformation matrix, however it is done in reverse, and with negative arguments, so it is actually the inverse of the transformation matrix. Then *lp* is created as a copy of the light's position, and multiplied by the matrix. Thus, *lp* is the light's position in the object's coordinate system.

```
glLoadIdentity(); // Reset Matrix
glRotatef(-yrot, 0.0f, 1.0f, 0.0f); // Rotate By -yrot On Y Axis
glRotatef(-xrot, 1.0f, 0.0f, 0.0f); // Rotate By -xrot On X Axis
glTranslatef(-ObjPos[0], -ObjPos[1], -ObjPos[2]); // Move Negative On All Axis Base
glGetFloatv(GL_MODELVIEW_MATRIX,Minv); // Retrieve ModelView Matrix (Stores :
lp[0] = LightPos[0]; // Store Light Position X In lp[0]
lp[1] = LightPos[1]; // Store Light Position Y In lp[1]
lp[2] = LightPos[2]; // Store Light Position Z In lp[2]
lp[3] = LightPos[3]; // Store Light Direction In lp[3]
VMatMult(Minv, lp); // We Store Rotated Light Vector In 'lp' Array
```

Now, palm off some of the work to draw the room, and the object. Calling **castShadow** draws the shadow of the object.

```
glLoadIdentity(); // Reset Modelview Matrix
glTranslatef(0.0f, 0.0f, -20.0f); // Zoom Into The Screen 20 Units
DrawGLRoom(); // Draw The Room
glTranslatef(ObjPos[0], ObjPos[1], ObjPos[2]); // Position The Object
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Spin It On The X Axis By xrot
glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Spin It On The Y Axis By yrot
drawObject(obj); // Procedure For Drawing The Loaded Object
castShadow(obj, lp); // Procedure For Casting The Shadow Based On The §
```

The following few lines draw a little orange circle where the light is:

The last part updates the object's position and returns.

```
xrot += xspeed; // Increase xrot By xspeed
yrot += yspeed; // Increase yrot By yspeed
glFlush(); // Flush The OpenGL Pipeline
return TRUE; // Everything Went OK
}
```

We did specify a **DrawGLRoom** function, and here it is - a bunch of rectangles to cast shadows against:

```
void DrawGLRoom()
                                                              // Draw The Room (Box)
{
    glBegin(GL_QUADS);
                                                               // Begin Drawing Quads
         // Floor
         glNormal3f(0.0f, 1.0f, 0.0f);
                                                                            // Normal Pointing Up
                                                                           // Back Left
         glVertex3f(-10.0f,-10.0f,-20.0f);
glVertex3f(-10.0f,-10.0f, 20.0f);
                                                                             // Front Left
                                                                             // Front Right
         glVertex3f( 10.0f,-10.0f, 20.0f);
         glVertex3f( 10.0f, -10.0f, -20.0f);
                                                                              // Back Right
         // Ceiling
         glNormal3f(0.0f,-1.0f, 0.0f);
                                                                           // Normal Point Down
                                                                           // Front Left
         glVertex3f(-10.0f, 10.0f, 20.0f);
                                                                             // Back Left
         glVertex3f(-10.0f, 10.0f,-20.0f);
         glVertex3f( 10.0f, 10.0f,-20.0f);
glVertex3f( 10.0f, 10.0f, 20.0f);
                                                                             // Back Right
// Front Right

      glvertex3f(-10.0f, 0.0f, 1.0f);
      // Normal Pointing Away From Viewer

      glvertex3f(-10.0f, 10.0f, -20.0f);
      // Top Left

      glvertex3f(10.0f, -10.0f, -20.0f);
      // Bottom Left

      glvertex3f(10.0f, 10.0f, -20.0f);
      // Bottom Right

      glvertex3f(10.0f, 10.0f, -20.0f);
      // Top Right

         glVertex3f( 10.0f, -1.0f); // Normal Pointing Towards Viewer
glVertex3f( 10.0f, 10.0f, 20.0f); // Top Right
glVertex3f( -10.0f, -10.0f, 20.0f); // Bottom Right
glVertex3f(-10.0f, 10.0f, 20.0f); // Bottom Left
         glVertex3f(-10.0f, 10.0f, 20.0f);
         // Left Wall
         glNormal3f(1.0f, 0.0f, 0.0f); // Normal Pointing Right
glVertex3f(-10.0f, 10.0f, 20.0f); // Top Front
glVertex3f(-10.0f,-10.0f, 20.0f); // Bottom Front
glVertex3f(-10.0f,-10.0f,-20.0f); // Bottom Back
         glVertex3f(-10.0f,-10.0f,-20.0f);
         glVertex3f(-10.0f, 10.0f,-20.0f);
                                                                              // Top Back
         // Right Wall
                                                                            // Normal Pointing Left
         glNormal3f(-1.0f, 0.0f, 0.0f);
                                                                           // Top Back
// Bottom Back
         glVertex3f( 10.0f, 10.0f, -20.0f);
glVertex3f( 10.0f, -10.0f, -20.0f);
         glVertex3f( 10.0f, -10.0f, 20.0f); // Bottom Front
glVertex3f( 10.0f, 10.0f, 20.0f); // Top Front
    glEnd();
                                                    // Done Drawing Quads
}
```

And before I forget, here is the **VMatMult** function which multiplies a vector by a matrix (get that Math textbook out again!):

```
v[0]=res[0]; // Results Are Stored Back In v[]
v[1]=res[1];
v[2]=res[2];
v[3]=res[3]; // Homogenous Coordinate
}
```

The function to load the object is simple, just calling **readObject**, and then setting up the connectivity and the plane equations for each face.

```
int InitGLObjects() // Initialize Objects
{
    if (!readObject("Data/Object2.txt", obj)) // Read Object2 Into obj
    {
        return FALSE; // If Failed Return False
    }
    setConnectivity(obj); // Set Face To Face Connectivity
    for ( int i=0;i < obj.nFaces;i++) // Loop Through All Object Faces
        calculatePlane(obj, obj.pFaces[i]); // Compute Plane Equations For All Faces
    return TRUE; // Return True
}</pre>
```

Finally, **KillGLObjects** is a convenience function so that if you add more objects, you can add them in a central place.

```
void KillGLObjects()
{
    killObject( obj );
}
```

All of the other functions don't require any further explanantion. I have left out the standard NeHe tutorial code, as well as all of the variable definitions and the keyboard processing function. The commenting alone explains these sufficiently.

Some things to note about the tutorial:

- The sphere doesn't stop shadows being projected on the wall. In reality, the sphere should also be casting a shadow, so seeing the one on the wall won't matter, it's hidden. It's just there to see what happens on curved surfaces :)
- If you are noticing extremely slow frame rates, try switching to fullscreen mode, or setting your desktop colour depth to 32bpp.
- Arseny L. writes: If you are having problems with a TNT2 in Windowed mode, make sure your desktop color depth is not set to 16bit. In 16bit color mode, the stencil buffer is emulated, resulting in sluggish performance. There are no problems in 32bit mode (I have a TNT2 Ultra and I checked it).

I've got to admit this was a lengthy task to write out this tutorial. It gives you full appreciation for the work that Jeff puts in! I hope you enjoy it, and give a huge thanks to Banu who wrote the original code! IF there is anything that needs further explaining in here, you are welcome to contact me (Brett), at brettporter@yahoo.com.

Banu Cosmin (Choko) - Original Code Brett Porter - HTML / Code Modifications Jeff Molofee (NeHe) - HTML Clean Up / Base Code

* DOWNLOAD Visual C++ Code For This Lesson.

* DOWNLOAD Mac OS Code For This Lesson. (Conversion by Morgan Aldridge)

Lesson 29

Bezier Patches

Written by: David Nikdel (ogapo@ithink.net)

This tutorial is intended to introduce you to Bezier Surfaces in the hopes that someone more artistic than myself will do something really cool with them and show all of us. This is not intended as a complete Bezier patch library, but more as proof of concept code to get you familiar with how these curved surfaces actually work. Also, as this is a very informal piece, I may have occasional lapses in correct terminology in favor of comprehensability; I hope this sits well with everyone. Finally, to those of you already familiar with Beziers who are just reading this to see if I screw up, shame on you ;-), but if you find anything wrong by all means let me or NeHe know, after all no one's perfect, eh? Oh, and one more thing, none of this code is optimised beyond my normal programming technique, this is by design. I want everyone to be able to see exactly what is going on. Well, I guess that's enough of an intro. On with the show!

The Math - ::evil music:: (warning, kinda long section)

Ok, it will be very hard to understand Beziers without at least a basic understanding of the math behind it, however, if you just don't feel like reading this section or already know the math, you can skip it. First I will start out by describing the Bezier curve itself then move on to how to create a Bezier Patch.

Odds are, if you've ever used a graphics program you are already familiar with Bezier curves, perhaps not by that name though. They are the primary method of drawing curved lines and are commonly represented as a series of points each with 2 points representing the tangent at that point from the left and right. Here's what one looks like:



This is the most basic Bezier curve possible (longer ones are made by attaching many of these together (many times without the user realizing it)). This curve is actually defined by only 4 points, those would be the 2 ending control points and the 2 middle control points. To the computer, all the points are the same, but to aid in design we often connect the first and the last two, respectively, because those lines will always be tangent to the endpoint. The curve is a parametric curve and is drawn by finding any number of points evenly spaced along the curve and connecting them with straight lines. In this way you can control the resolution of the patch (and the amount of computation). The most common way to use this is to tesselate it less at a farther distance and more at a closer distance so that, to the viewer, it always appears to be a perfectly curved surface with the lowest possible speed hit.

Bezier curves are based on a basis function from which more complicated versions are derived. Here's the function:

t + (1 - t) = 1

Sounds simple enough huh? Well it really is, this is the Bezier most basic Bezier curve, a 1st degree curve. As you may have guessed from the terminology, the Bezier curves are polynomials, and as we remember from algebra, a 1st degree polynomial is just a straight line; not very interesting. Well, since the basis function is true for all numbers t, we can square, cube, whatever, each side and it will still be true right? Well, lets try cubing it.

 $(t + (1 - t))^3 = 1^3$

$t^3 + 3^{t^2}(1-t) + 3^{t^*}(1-t)^2 + (1-t)^3 = 1$

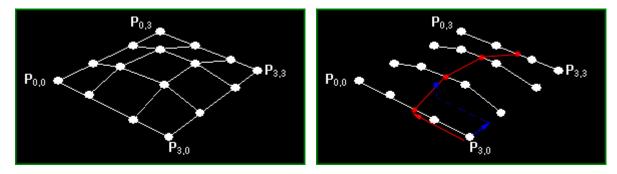
This is the equation we use to calculate the most common Bezier, the 3rd degree Bezier curve (yes, it's a strange phenomenon, but sometimes when you're doing math the functions just come out all rainbow colored ; -)). This is most common for two reasons, a) it's the lowest degree polynomial that need not necessarily lie in a plane (there are 4 control points) and b) the tangent lines on the sides are not dependant on one another (with a 2nd degree there would be only 3 control points). So do you see the Bezier curve yet? Hehe, me neither, that's because I still need to add one thing.

Ok, since the entire left side is equal to 1, it's safe to assume that if you add all the components they should still equal one. Does this sound like it could be used to descide how much of each control point to use in calculating a point on the curve? (hint: just say yes ;-)) Well you're right! When we want to calculate the value of a point some percent along the curve we simply multiply each part by a control point (as a vector) and find the sum. Generally, we'll work with $0 \le t \le 1$, but it's not technically necesary. Confused yet? Here's the function:

$P1*t^{3} + P2*3*t^{2}(1-t) + P3*3*t^{(1-t)} + P4*(1-t)^{3} = 1$

Because polynomials are always continuous, this makes for a good way to morp between the 4 points. The only points it actually reaches though are P1 and P4, when t = 1 and 0 respectively.

Now, that's all well and good, but how can I use these in 3D you ask? Well it's actually quite simple, in order to form a Bezier patch, you need 16 control points (4*4), and 2 variables \mathbf{t} and \mathbf{v} . What you do from there is calculate a point at \mathbf{v} along 4 of the parallel curves then use those 4 points to make a new curve and calculate \mathbf{t} along that curve. By calculating enough of these points, we can draw triangle strips to connect them, thus drawing the Bezier patch.



Well, I suppose that's enough math for now, on to the code!

```
#include <windows.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <gl\gl.h>
#include <gl\glaux.h>

typedef struct point_3d {
   double x, y, z;
} POINT_3D;

typedef struct bpatch {
   POINT_3D anchors[4][4];
}
```

dlBPatch;

GLuint

```
// Header File For Windows
// Header File For Math Library Routines
// Header File For Standard I/O Routines
// Header File For Standard Library
// Header File For The OpenGL32 Library
// Header File For The GLu32 Library
// Header File For The Glaux Library
// Structure For A 3-Dimensional Point ( NEW )
```

// Structure For A 3rd Degree Bezier Patch (NI
 // 4x4 Grid Of Anchor Points
// Display List For Bezier Patch

```
// Texture For The Patch
  GLuint
           texture;
} BEZIER_PATCH;
                                // Private GDI Device Context
          hDC=NULL;
HDC
HGLRC
           hRC=NULL;
                                  // Permanent Rendering Context
          hWnd=NULL;
HWND
                                   // Holds Our Window Handle
HINSTANCE
                                      // Holds The Instance Of The Application
             hInstance;
DEVMODE
             DMsaved;
                                    // Saves The Previous Screen Settings ( NEW )
     keys[256];
active=TRUE;
                                  // Array Used For The Keyboard Routine
bool
                                    // Window Active Flag Set To TRUE By Default
bool
bool
           fullscreen=TRUE;
                                      // Fullscreen Flag Set To Fullscreen Mode By Defau
             rotz = 0.0f;
GLfloat
                                         // Rotation About The Z Axis
BEZIER_PATCH
                                        // The Bezier Patch We're Going To Use ( NEW )
              mybezier;
BOOL showCPoints=TRUE;
                                        // Toggles Displaying The Control Point Grid ( NI
          divs = 7;
                                 // Number Of Intrapolations (Controls Poly Resolution)
int
```

The following are just a few quick functions for some simple vector math. If you're a fan of C++ you

might consider using a point class (just make sure it's 3d).

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

```
// Adds 2 Points. Don't Just Use '+' ;)
POINT_3D pointAdd(POINT_3D p, POINT_3D q) {
  p.x += q.x; p.y += q.y; p.z += q.z;
  return p;
}
// Multiplies A Point And A Constant. Don't Just Use '*'
POINT_3D pointTimes(double c, POINT_3D p) {
  p.x *= c; p.y *= c; p.z *= c;
  return p;
}
// Function For Quick Point Creation
POINT_3D makePoint(double a, double b, double c) {
  POINT_3D p;
  p.x = a; p.y = b; p.z = c;
  return p;
}
```

This is basically just the 3rd degree basis function written in C, it takes a variable \mathbf{u} and an array of 4 points and computes a point on the curve. By stepping \mathbf{u} in equal increments between 0 and 1, we'll get a nice approximation of the curve.

```
// Calculates 3rd Degree Polynomial Based On Array Of 4 Points
// And A Single Variable (u) Which Is Generally Between 0 And 1
POINT_3D Bernstein(float u, POINT_3D *p) {
    POINT_3D a, b, c, d, r;
    a = pointTimes(pow(u,3), p[0]);
    b = pointTimes(3*pow(u,2)*(1-u), p[1]);
    c = pointTimes(3*u*pow((1-u),2), p[2]);
    d = pointTimes(pow((1-u),3), p[3]);
    r = pointAdd(pointAdd(a, b), pointAdd(c, d));
    return r;
}
```

// Declaration For WndProc

This function does the lion's share of the work by generating all the triangle strips and storing them in a display list. We do this so that we don't have to recalculate the patch each frame, only when it changes. By the way, a cool effect you might want to try might be to use the morphing tutorial to morph the patch's control points. This would yeild a very cool smooth, organic, morphing effect for relatively little overhead (you only morph 16 points, but you have to recalculate). The "last" array is used to keep the previous line of points (since a triangle strip needs both rows). Also, texture coordinates are calculated by using the u and v values as the percentages (planar mapping).

One thing we don't do is calculate the normals for lighting. When it comes to this, you basically have two options. The first is to find the center of each triangle, then use a bit of calculus and calculate the tangent on both the x and y axes, then do the cross product to get a vector perpendicular to both, THEN normalize the vector and use that as the normal. OR (yes, there is a faster way) you can cheat and just use the normal of the triangle (calculated your favorite way) to get a pretty good approximation. I prefer the latter; the speed hit, in my opinion, isn't worth the extra little bit of realism.

```
// Generates A Display List Based On The Data In The Patch
// And The Number Of Divisions
GLuint genBezier(BEZIER_PATCH patch, int divs) {
  int u = 0, v;
  float py, px, pyold;
GLuint drawlist = glo
            drawlist = glGenLists(1);
                                            // Make The Display List
  POINT_3D temp[4];
  POINT_3D *last = (POINT_3D*)malloc(sizeof(POINT_3D)*(divs+1));
           // Array Of Points To Mark The First Line Of Polys
                                         // Get Rid Of Any Old Display Lists
  if (patch.dlBPatch != NULL)
     glDeleteLists(patch.dlBPatch, 1);
  temp[0] = patch.anchors[0][3];
                                           // The First Derived Curve (Along X-Axis)
  temp[1] = patch.anchors[1][3];
  temp[2] = patch.anchors[2][3];
  temp[3] = patch.anchors[3][3];
    r (v=0;v<=divs;v++) { // Create The First Line Of Points
px = ((float)v)/((float)divs); // Percent Along Y-Axis
  for (v=0;v<=divs;v++) {</pre>
   // Use The 4 Points From The Derived Curve To Calculate The Points Along That Curve
     last[v] = Bernstein(px, temp);
   }
  glNewList(drawlist, GL_COMPILE); // Start A New Display List
  glBindTexture(GL_TEXTURE_2D, patch.texture); // Bind The Texture
  for (u=1;u<=divs;u++) {</pre>
     py = ((float)u)/((float)divs); // Percent Along Y-Axis
pyold = ((float)u-1.0f)/((float)divs); // Percent Along 0.
                                                 // Percent Along Old Y Axis
     temp[1] = Bernstein(py, patch.anchors[1]);
     temp[2] = Bernstein(py, patch.anchors[2]);
     temp[3] = Bernstein(py, patch.anchors[3]);
     glBegin(GL_TRIANGLE_STRIP); // Begin A New Triangle Strip
     for (v=0;v<=divs;v++) {</pre>
        px = ((float)v)/((float)divs);
                                            // Percent Along The X-Axis
                                 // Apply The Old Texture Coords
        glTexCoord2f(pyold, px);
        glVertex3d(last[v].x, last[v].y, last[v].z); // Old Point
        glVertex3d(last[v].x, last[v].y, last[v].z); // New Point
     }
     glEnd(); // END The Triangle Strip
  }
```

```
glEndList();// END The Listfree(last);// Free The Old Vertices Arrayreturn drawlist;// Return The Display List
```

Here we're just loading the matrix with some values I've picked that I think look cool. Feel free to screw around with these and see what it looks like. :-)

```
void initBezier(void) {
  mybezier.anchors[0][0] = makePoint(-0.75, -0.75, -0.50);
                                                               // Set The Bezier Vertice
  mybezier.anchors[0][1] = makePoint(-0.25, -0.75, 0.00);
  mybezier.anchors[0][2] = makePoint( 0.25, -0.75,
                                                      0.00);
  mybezier.anchors[0][3] = makePoint( 0.75,
                                            -0.75,
                                                     -0.50);
  mybezier.anchors[1][0] = makePoint(-0.75,
                                            -0.25,
                                                     -0.75);
                                            -0.25,
  mybezier.anchors[1][1] = makePoint(-0.25,
                                                     0.50);
  mybezier.anchors[1][2] = makePoint( 0.25, -0.25,
                                                     0.50);
  mybezier.anchors[1][3] = makePoint( 0.75,
                                           -0.25,
                                                    -0.75);
                                           0.25,
0.25,
                                                     0.00);
  mybezier.anchors[2][0] = makePoint(-0.75,
  mybezier.anchors[2][1] = makePoint(-0.25,
                                                     -0.50);
                                             0.25,
  mybezier.anchors[2][2] = makePoint( 0.25,
                                                     -0.50);
  mybezier.anchors[2][3] = makePoint( 0.75,
                                            0.25,
                                                     0.00);
  mybezier.anchors[3][0] = makePoint(-0.75,
                                           0.75,
                                                    -0.50);
  mybezier.anchors[3][1] = makePoint(-0.25,
                                           0.75,
                                                     -1.00);
                                           0.75,
  mybezier.anchors[3][2] = makePoint( 0.25,
                                                     -1.00);
  mybezier.anchors[3][3] = makePoint( 0.75,
                                                     -0.50);
  mybezier.dlBPatch = NULL;
                                         // Go Ahead And Initialize This To NULL
}
```

This is basically just an optimised routine to load a single bitmap. It can easily be used to load an array of em just by putting it in a simple loop.

```
// Load Bitmaps And Convert To Textures
BOOL LoadGLTexture(GLuint *texPntr, char* name)
ł
  BOOL success = FALSE;
  AUX_RGBImageRec *TextureImage = NULL;
  glGenTextures(1, texPntr); // Generate 1 Texture
  FILE* test=NULL;
  TextureImage = NULL;
  test = fopen(name, "r");
                                        // Test To See If The File Exists
  if (test != NULL) {
                                      // If It Does
     fclose(test);
                                   // Close The File
                                            // And Load The Texture
     TextureImage = auxDIBImageLoad(name);
   }
                                         // If It Loaded
   if (TextureImage != NULL) {
     success = TRUE;
      // Typical Texture Generation Using Data From The Bitmap
      glBindTexture(GL_TEXTURE_2D, *texPntr);
     glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage->sizeX, TextureImage->sizeY, 0, GL_RGH
     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
  }
  if (TextureImage->data)
      free(TextureImage->data);
```

```
return success;
```

}

Just adding the patch initialization here. You would do this whenever you create a patch. Again, this might be a cool place to use C++ (bezier class?).

```
int InitGL(GLvoid)
                                    // All Setup For OpenGL Goes Here
{
  glEnable(GL_TEXTURE_2D);
                                        // Enable Texture Mapping
  glShadeModel(GL_SMOOTH);
                                        // Enable Smooth Shading
  glClearColor(0.05f, 0.05f, 0.05f, 0.5f);
                                          // Black Background
  glClearDepth(1.0f);
                                    // Depth Buffer Setup
  glEnable(GL_DEPTH_TEST);
                                       // Enables Depth Testing
  glDepthFunc(GL_LEQUAL);
                                         // The Type Of Depth Testing To Do
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
                                                        // Really Nice Perspective Calcu
                                  // Initialize the Bezier's Control Grid ( NEW )
  initBezier();
  LoadGLTexture(&(mybezier.texture), "./Data/NeHe.bmp"); // Load The Texture ( NEW )
  mybezier.dlBPatch = genBezier(mybezier, divs); // Generate The Patch ( NEW )
                                  // Initialization Went OK
  return TRUE;
}
```

First call the bezier's display list. Then (if the outlines are on) draw the lines connecting the control points. You can toggle these by pressing SPACE.

```
int DrawGLScene(GLvoid) {
                                            // Here's Where We Do All The Drawing
  int i, j;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                                            // Clear Screen And Depth Buffe
  glLoadIdentity();
                                     // Reset The Current Modelview Matrix
  glTranslatef(0.0f,0.0f,-4.0f);
                                                // Move Left 1.5 Units And Into The Screen
  glRotatef(-75.0f,1.0f,0.0f,0.0f);
  glRotatef(rotz,0.0f,0.0f,1.0f);
                                                 // Rotate The Triangle On The Z-Axis
                                                // Call The Bezier's Display List
  glCallList(mybezier.dlBPatch);
                          // This Need Only Be Updated When The Patch Changes
   if (showCPoints) {
                                       // If Drawing The Grid Is Toggled On
     glDisable(GL_TEXTURE_2D);
     glColor3f(1.0f,0.0f,0.0f);
     for(i=0;i<4;i++) {</pre>
                                       // Draw The Horizontal Lines
        glBegin(GL_LINE_STRIP);
        for(j=0;j<4;j++)</pre>
            glVertex3d(mybezier.anchors[i][j].x, mybezier.anchors[i][j].y, mybezier.anchors
        glEnd();
      }
      for(i=0;i<4;i++) {
                                      // Draw The Vertical Lines
        glBegin(GL_LINE_STRIP);
        for(j=0;j<4;j++)</pre>
           glVertex3d(mybezier.anchors[j][i].x, mybezier.anchors[j][i].y, mybezier.anchors
        qlEnd();
     glColor3f(1.0f,1.0f,1.0f);
     glEnable(GL_TEXTURE_2D);
   }
  return TRUE;
                                   // Keep Going
}
```

This function contains some modified code to make your projects more compatable. It doesn't have anything to do with Bezier curves, but it does fix a problem with switching back the resolution after

fullscreen mode with some video cards (including mine, a crappy old ATI Rage PRO, and a few others). I hope, you'll use this from now on so me and others with similar cards can view your cool examples GL code properly. To make these modifications make the changes in KillGLWindow(), make sure and define DMsaved, and make the one line change in CreateGLWindow() (it's marked).

```
GLvoid KillGLWindow(GLvoid)
                                             // Properly Kill The Window
                                      // Are We In Fullscreen Mode?
   if (fullscreen)
   {
      if (!ChangeDisplaySettings(NULL,CDS_TEST)) {
                                                         // If The Shortcut Doesn't Work (
                                                    // Do It Anyway (To Get The Values Out
         ChangeDisplaySettings(NULL,CDS_RESET);
         ChangeDisplaySettings(&DMsaved,CDS_RESET); // Change It To The Saved Settings (
      } else {
         ChangeDisplaySettings(NULL,CDS_RESET);
                                                    // If It Works, Go Right Ahead ( NEW
      }
      ShowCursor(TRUE);
                                     // Show Mouse Pointer
   }
   if (hRC)
                                // Do We Have A Rendering Context?
   ł
      if (!wglMakeCurrent(NULL,NULL))
                                                 // Are We Able To Release The DC And RC Co
      ł
        MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR", MB_OK | MB_ICONINI
      }
      if (!wglDeleteContext(hRC))
                                            // Are We Able To Delete The RC?
      ł
        MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN ERROR", MB_OK | MB_I(
     hRC=NULL;
                                // Set RC To NULL
   }
   if (hDC && !ReleaseDC(hWnd,hDC))
                                              // Are We Able To Release The DC
   ł
     MessageBox(NULL, "Release Device Context Failed.", "SHUTDOWN ERROR", MB_OK | MB_ICONINF(
     hDC=NULL;
                                // Set DC To NULL
   }
   if (hWnd && !DestroyWindow(hWnd))
                                               // Are We Able To Destroy The Window?
   {
     MessageBox(NULL, "Could Not Release hWnd.", "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION
     hWnd=NULL;
                                 // Set hWnd To NULL
   }
   if (!UnregisterClass("OpenGL",hInstance))
                                                    // Are We Able To Unregister Class
   {
      MessageBox(NULL, "Could Not Unregister Class.", "SHUTDOWN ERROR", MB_OK | MB_ICONINFORM
     hInstance=NULL;
                                      // Set hInstance To NULL
}
```

Just added the EnumDisplaySettings() command here to save the old display settings. (part of the old graphics card fix).

/*	This Code Creates Our OpenGL Window. Parameters Are: *
*	title - Title To Appear At The Top Of The Window *
*	width - Width Of The GL Window Or Fullscreen Mode *
*	height - Height Of The GL Window Or Fullscreen Mode *
*	bits - Number Of Bits To Use For Color (8/16/24/32) *
*	fullscreenflag – Use Fullscreen Mode (TRUE) Or Windowed Mode (FALSE) */
BOOL	CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)

{

}

```
PixelFormat;
GLuint
                                                                                            // Holds The Results After Searching For A Match
GLuintPixelFormat;// Holds The Results After Searching For A MatchWNDCLASSwc;// Windows Class StructureDWORDdwExStyle;// Window Extended StyleDWORDdwStyle;// Window StyleRECTWindowRect;// Grabs Rectangle Upper Left / Lower Right ValuesWindowRect.left=(long)0;// Set Left Value To 0WindowRect.top=(long)0;// Set Top Value To 0WindowRect.bottom=(long)height;// Set Bottom Value To Requested Height
fullscreen=fullscreenflag;
                                                                                               // Set The Global Fullscreen Flag
                                = GetModuleHandle(NULL); // Grab An Instance For Our Window
hInstance
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // Redraw On Size, And Own DC For
wc.lpfnWndProc = (WNDPROC) WndProc; // WndProc Handles Messages
wc.cbClsExtra = 0; // No Extra Window Data
wc.cbWndExtra = 0; // No Extra Window Data
wc.hInstance = hInstance; // Set The Instance
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Load The Default Icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Load The Arrow Pointer
wc.hbrBackground = NULL; // No Background Required For GL
wc.lpszMenuName = NULL; // We Don't Want A Menu
wc.lpszClassName = "OpenGL"; // Set The Class Name
                                                                                              // We Don C ....
// Set The Class Name
EnumDisplaySettings(NULL, ENUM_CURRENT_SETTINGS, &DMsaved); // Save The Current DisplaySettings(NULL, ENUM_CURRENT_SETTINGS, ENUM_CURRENT_SET
 if (fullscreen)
                                                                                   // Attempt Fullscreen Mode?
 {
       DEVMODE dmScreenSettings;
                                                                                             // Device Mode
       memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's Cleare
       dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode Structure
       dmScreenSettings.dmPelsWidth = width; // Selected Screen Width
dmScreenSettings.dmPelsHeight = height; // Selected Screen Height
dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pixel
       dmScreenSettings.dmFields=DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;
 ... Code Cut To Save Space (No Further Changes To This Function) ...
return TRUE;
                                                                            // Success
```

All I did here was add commands to rotate the patch, raise/lower the resolution, and toggle the control lines.

```
int WINAPI WinMain( HINSTANCE hInstance,
                                                    // Instance
              HINSTANCE hPrevInstance, // Previous Instance
              LPSTR lpCmdLine, // Commany _
int nCmdShow) // Window Show State
                         lpCmdLine, // Command Line Parameters
{
  MSG msg;
BOOL done=FALSE;
                                 // Windows Message Structure
                                      // Bool Variable To Exit Loop
   // Ask The User Which Screen Mode They Prefer
  if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreen?", MB
  {
     fullscreen=FALSE;
                                     // Windowed Mode
  }
   // Create Our OpenGL Window
  if (!CreateGLWindow("NeHe's Solid Object Tutorial",640,480,16,fullscreen))
   {
     return 0;
                                // Quit If Window Was Not Created
   }
  while(!done)
                                    // Loop That Runs While done=FALSE
     if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
```

}

}

```
{
      if (msg.message==WM_QUIT)
                                        // Have We Received A Quit Message?
      {
                              // If So done=TRUE
         done=TRUE;
      }
      else
                           // If Not, Deal With Window Messages
      {
         TranslateMessage(&msg);
                                         // Translate The Message
                                      // Dispatch The Message
         DispatchMessage(&msg);
      }
                            // If There Are No Messages
   else
   {
      // Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
      if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit
      {
        done=TRUE;
                               // ESC or DrawGLScene Signalled A Quit
      }
                           // Not Time To Quit, Update Screen
      else
      {
         SwapBuffers(hDC);
                                  // Swap Buffers (Double Buffering)
      }
      if (keys[VK_LEFT]) rotz -= 0.8f; // Rotate Left ( NEW )
if (keys[VK_RIGHT]) rotz += 0.8f; // Rotate Right ( NEW )
      if (keys[VK_UP]) {
                                   // Resolution Up ( NEW )
         divs++;
         mybezier.dlBPatch = genBezier(mybezier, divs); // Update The Patch
         keys[VK_UP] = FALSE;
      if (keys[VK_DOWN] && divs > 1) { // Resolution Down ( NEW )
         divs--;
         mybezier.dlBPatch = genBezier(mybezier, divs); // Update The Patch
         keys[VK_DOWN] = FALSE;
                                      // SPACE Toggles showCPoints ( NEW )
      if (keys[VK_SPACE]) {
         showCPoints = !showCPoints;
         keys[VK_SPACE] = FALSE;
      }
      if (keys[VK_F1]) // Is F1 Being Pressed?
      {
        keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Win
                                   // Kill Our Current Window
         fullscreen=!fullscreen;
                                         // Toggle Fullscreen / Windowed Mode
         // Recreate Our OpenGL Window
         if (!CreateGLWindow("NeHe's Solid Object Tutorial",640,480,16,fullscreen))
         {
                             // Quit If Window Was Not Created
            return 0;
         }
      }
   }
// Shutdown
KillGLWindow();
                                   // Kill The Window
return (msg.wParam);
                                     // Exit The Program
```

Well, I hope this tutorial has been enlightening and you all now love Bezier curves as much as I do ;-). If you like this tutorial I may write another one on NURBS curves if anyone's interested. Please email me and let me know what you thought of this tutorial.

About The Author: David Nikdel is currently 18 and a senior at Bartow Senior High School. His current projects include a research paper on curved surfaces in 3D graphics, an OpenGL based game called Blazing Sands and being lazy. His hobbies include programming, football, and paintballing. He will (hopefully) be a freshman at Georgia Tech next year.

David Nikdel - Code Jeff Molofee (NeHe) - HTML

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Visual C++ / OpenIL Code For This Lesson. (Conversion by Denton Woods)
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Morgan Aldridge)
- * DOWNLOAD Irix / GLUT Code For This Lesson. (Conversion by Rob Fletcher)
- * DOWNLOAD Delphi Code For This Lesson. (Conversion by Steven Brom)
- * DOWNLOAD MacOS X / GLUT Code For This Lesson. (Conversion by Ben Reichardt)

Lesson 30

This tutorial was originally written by Andreas Löffler. He also wrote all of the original HTML for the tutorial. A few days later Rob Fletcher emailed me an Irix version of lesson 30. In his version he rewrote most of the code. So I ported Rob's Irix / GLUT code to Visual C++ / Win32. I then modified the message loop code, and the fullscreen code. When the program is minimized it should use 0% of the CPU (or close to). When switching to and from fullscreen mode, most of the problems should be gone (screen not restoring properly, messed up display, etc).

Andreas tutorial is now better than ever. Unfortunately, the code has been modifed quite a bit, so all of the HTML has been rewritten by myself. Huge Thanks to Andreas for getting the ball rolling, and working his butt off to make a killer tutorial. Thanks to Rob for the modifications!

Lets begin... We create a device mode structure called **DMsaved**. We will use this structure to store information about the users default desktop resolution, color depth, etc., before we switch to fullscreen mode. More on this later! Notice we only allocate enough storage space for one texture (texture[1]).

#include	<windows.h> <gl\gl.h> <gl\glu.h> <stdio.h></stdio.h></gl\glu.h></gl\gl.h></windows.h>	<pre>// Header File For Windows // Header File For The OpenGL32 Library // Header File For The GLu32 Library // Header File For File Operation Needed</pre>
HDC HGLRC HWND HINSTANCE	hRC=NULL; hWnd=NULL;	<pre>// Private GDI Device Context // Permanent Rendering Context // Holds Our Window Handle</pre>
	<pre>keys[256]; active=TRUE; fullscreen=TRUE;</pre>	<pre>// Array Used For The Keyboard Routine // Window Active Flag Set To TRUE By Default // Fullscreen Flag Set To Fullscreen Mode By</pre>
DEVMODE	DMsaved;	// Saves The Previous Screen Settings (NEW)
GLfloat GLfloat GLfloat	<pre>xrot; yrot; zrot;</pre>	<pre>// X Rotation // Y Rotation // Z Rotation</pre>
GLuint	<pre>texture[1];</pre>	// Storage For 1 Texture

Now for the fun stuff. We create a structure called **TEXTURE_IMAGE**. The structure contains information about our images **width**, **height**, and **format** (bytes per pixel). **data** is a pointer to unsigned char. Later on data will point to our image data.

We then create a pointer called **P_TEXTURE_IMAGE** to the **TEXTURE_IMAGE** data type. The variables **t1** and **t2** are of type **P_TEXTURE_IMAGE** where **P_TEXTURE_IMAGE** is a redefined type of pointer to **TEXTURE_IMAGE**.

typedef TEXTURE_IMAGE *P_TEXTURE_IMAGE;	// A Pointer To The Texture Ima
P_TEXTURE_IMAGE t1; P_TEXTURE_IMAGE t2;	// Pointer To The Texture Image Data Type // Pointer To The Texture Image Data Type
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM	I, LPARAM); // Declaration For Wi

Below is the code to allocate memory for a texture. When we call this code, we pass it the width, height and bytes per pixel information of the image we plan to load. **ti** is a pointer to our **TEXTURE_IMAGE** data type. It's given a NULL value. **c** is a pointer to unsigned char, it is also set to NULL.

```
// Allocate An Image Structure And Inside Allocate Its Memory Requirements
P_TEXTURE_IMAGE AllocateTextureBuffer( GLint w, GLint h, GLint f)
{
    P_TEXTURE_IMAGE ti=NULL; // Pointer To Image Struct
    unsigned char *c=NULL; // Pointer To Block Memory For Image
```

Here is where we allocate the memory for our image structure. If everything goes well, ti will point to the allocated memory.

After allocating the memory, and checking to make sure ti is not equal to NULL, we can fill the structure with the image attributes. First we set the width (w), then the height (h) and lastly the format (f). Keep in mind format is bytes per pixel.

```
ti = (P_TEXTURE_IMAGE)malloc(sizeof(TEXTURE_IMAGE)); // One Image Struct Ple;
if( ti != NULL ) {
  ti->width = w; // Set Width
  ti->height = h; // Set Height
  ti->format = f; // Set Format
```

Now we need to allocate memory for the actual image data. The calculation is easy! We multiply the width of the image (w) by the height of the image (h) then multiply by the format (f - bytes per pixel).

c = (unsigned char *)malloc(w * h * f);

We check to see if everything went ok. If the value in **c** is not equal to NULL we set the **data** variable in our structure to point to the newly allocated memory.

If there was a problem, we pop up an error message on the screen letting the user know that the program was unable to allocate memory for the texture buffer. NULL is returned.

```
if ( c != NULL ) {
    ti->data = c;
}
```

```
else {
    MessageBox(NULL,"Could Not Allocate Memory For A Texture Buffer","BUFFER ERROR",Mi
    return NULL;
  }
}
```

If anything went wrong when we were trying to allocate memory for our image structure, the code below would pop up an error message and return NULL.

If there were no problems, we return **ti** which is a pointer to our newly allocated image structure. Whew... Hope that all made sense.

```
else
{
    MessageBox(NULL,"Could Not Allocate An Image Structure","IMAGE STRUCTURE ERROR",MB_OF
    return NULL;
}
return ti;
// Return Pointer To Image Struct
}
```

When it comes time to release the memory, the code below will deallocate the texture buffer and then free the image structure. **t** is a pointer to the **TEXTURE_IMAGE** data structure we want to deallocate.

```
// Free Up The Image Data
void DeallocateTexture( P_TEXTURE_IMAGE t )
{
    if (t->data)
    {
        free( t->data ); // Free Its Image Buffer
    }
    if (t)
    {
        free(t); // Free Itself
    }
}
```

Now we read in our .RAW image. We pass the **filename** and a pointer to the image structure we want to load the image into. We set up our misc variables, and then calculate the size of a row. We figure out the size of a row by multiplying the **width** of our image by the **format** (bytes per pixel). So if the image was 256 pixels wide and there were 4 bytes per pixel, the width of a row would be 1024 bytes. We store the width of a row in **stride**.

We set up a pointer (**p**), and then attempt to open the file.

```
// Read A .RAW File In To The Allocated Image Buffer Using data In The Image Structure Heac
// Flip The Image Top To Bottom. Returns 0 For Failure Of Read, Or Number Of Bytes Read.
int ReadTextureData ( char *filename, P_TEXTURE_IMAGE buffer)
{
    FILE *f;
    int i,j,k,done=0;
    int stride = buffer->width * buffer->format; // Size Of A Row (Width * Byt
    unsigned char *p = NULL;
    f = fopen(filename, "rb"); // Open "filename" For Reading Bytes
    if( f != NULL ) // If File Exists
    {
    }// Size Of A Row (Width * Duffer ->format; // Size Of A Row (Width * Duffer ->format; // Size Of A Row (Width * Duffer ->format; // Size Of A Row (Width * Byt
    unsigned char *p = NULL; // Open "filename" For Reading Bytes
    if( f != NULL ) // If File Exists // If File Exists
```

If the file exists, we set up the loops to read in our texture. **i** starts at the bottom of the image and moves up a line at a time. We start at the bottom so that the image is flipped the right way. RAW images are stored upside down. We have to set our pointer now so that the data is loaded into the proper spot in the image buffer. Each time we move up a line (**i** is decreased) we set the pointer to the start of the new line. **data** is where our image buffer starts, and to move an entire line at a time in the buffer, multiply **i** by **stride**. Remember that **stride** is the length of a line in bytes, and **i** is the current line. So by multiplying the two, we move an entire line at a time.

The **j** loop moves from left (0) to right (**width** of line in pixels, not bytes).

```
for( i = buffer->height-1; i >= 0 ; i-- ) // Loop Through Height (Bottoms
{
    p = buffer->data + (i * stride );
    for ( j = 0; j < buffer->width ; j++ ) // Loop Through Width
    {
```

The **k** loop reads in our bytes per pixel. So if **format** (bytes per pixel) is 4, **k** loops from 0 to 2 which is bytes per pixel minus one (**format-1**). The reason we subtract one is because most raw images don't have an alpha value. We want to make the 4th byte our alpha value, and we want to set the alpha value manually.

Notice in the loop we also increase the pointer (**p**) and a variable called **done**. More about **done** later.

the line inside the loop reads a character from our file and stores it in the texture buffer at our current pointer location. If our image has 4 bytes per pixel, the first 3 bytes will be read from the .RAW file (**format-1**), and the 4th byte will be manually set to 255. After we set the 4th byte to 255 we increase the pointer location by one so that our 4th byte is not overwritten with the next byte in the file.

After a all of the bytes have been read in per pixel, and all of the pixels have been read in per row, and all of the rows have been read in, we are done! We can close the file.

```
for ( k = 0 ; k < buffer->format-1 ; k++, p++, done++ )
{
            *p = fgetc(f); // Read Value From File And Store In Memory
            *p = 255; p++; // Store 255 In Alpha Channel And Increase Poir
            }
            fclose(f); // Close The File
}
```

If there was a problem opening the file (does not exist, etc), the code below will pop up a message box letting the user know that the file could not be opened.

The last thing we do is return **done**. If the file couldn't be opened, **done** will equal 0. If everything went ok, **done** should equal the number of bytes read from the file. Remember, we were increasing **done** every time we read a byte in the loop above (**k** loop).

```
else // Otherwise
{
    MessageBox(NULL,"Unable To Open Image File","IMAGE ERROR",MB_OK | MB_ICONINFORMATION
}
return done; // Returns Number Of Bytes Read In
```

}

This shouldn't need explaining. By now you should know how to build a texture. **tex** is the pointer to the **TEXTURE_IMAGE** structure that we want to use. We build a linear filtered texture. In this example, we're building mipmaps (smoother looking). We pass the **width**, **height** and **data** just like we would if we were using glaux, but this time we get the information from the selected **TEXTURE_IMAGE** structure.

```
void BuildTexture (P_TEXTURE_IMAGE tex)
{
    glGenTextures(1, &texture[0]);
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    gluBuild2DMipmaps(GL_TEXTURE_2D,GL_RGB, tex->width, tex->height, GL_RGBA, GL_UNSIGNED_F
}
```

Now for the blitter code :) The blitter code is very powerful. It lets you copy any section of a (**src**) texture and paste it into a destination (**dst**) texture. You can combine as many textures as you want, you can set the alpha value used for blending, and you can select whether the two images blend together or cancel eachother out.

src is the TEXTURE_IMAGE structure to use as the source image. dst is the TEXTURE_IMAGE
structure to use for the destination image. src_xstart is where you want to start copying from on the
x axis of the source image. src_ystart is where you want to start copying from on the y axis of the
source image. src_width is the width in pixels of the area you want to copy from the source image.
src_height is the height in pixels of the area you want to copy from the source image. dst_xstart
and dst_ystart is where you want to place the copied pixels from the source image onto the
destination image. If blend is 1, the two images will be blended. alpha sets how tranparent the
copied image will be when it mapped onto the destination image. 0 is completely clear, and 255 is
solid.

We set up all our misc loop variables, along with pointers for our source image (s) and destination image (d). We check to see if the **alpha** value is within range. If not, we clamp it. We do the same for the **blend** value. If it's not 0-off or 1-on, we clamp it.

```
void Blit( P_TEXTURE_IMAGE src, P_TEXTURE_IMAGE dst, int src_xstart, int src_ystart, int si
    int dst_xstart, int dst_ystart, int blend, int alpha)
{
    int i,j,k;
    unsigned char *s, *d; // Source & Destination
    // Clamp Alpha If Value Is Out Of Range
    if( alpha > 255 ) alpha = 255;
    if( alpha < 0 ) alpha = 0;
    // Check For Incorrect Blend Flag Values
    if( blend < 0 ) blend = 0;
    if( blend < 0 ) blend = 1;
    </pre>
```

Now we have to set up the pointers. The destination pointer is the location of the destination **data** plus the starting location on the destination images y axis (**dst_ystart**) * the destination images **width** in pixels * the destination images bytes per pixel (**format**). This should give us the starting row for our destination image.

We do pretty much the same thing for the source pointer. The source pointer is the location of the source **data** plus the starting location on the source images y axis (**src_ystart**) * the source images **width** in pixels * the source images bytes per pixel (**format**). This should give us the starting row for our source image.

i loops from 0 to **src_height** which is the number of pixels to copy up and down from the source image.

```
for (i = 0 ; i < src_height ; i++ )
                  // Height Loop
{
```

We already set the source and destination pointers to the correct rows in each image. Now we have to move to the correct location from left to right in each image before we can start blitting the data. We increase the location of the source pointer (s) by src_xstart which is the starting location on the x axis of the source image times the source images bytes per pixel. This moves the source (s) pointer to the starting pixel location on the x axis (from left to right) on the source image.

We do the exact same thing for the destination pointer. We increase the location of the destination pointer (d) by dst_xstart which is the starting location on the x axis of the destination image multiplied by the destination images bytes per pixel (format). This moves the destination (d) pointer to the starting pixel location on the x axis (from left to right) on the destination image.

After we have calculated where in memory we want to grab our pixels from (s) and where we want to move them to (d), we start the j loop. We'll use the j loop to travel from left to right through the source image.

{

}

The k loop is used to go through all the bytes per pixel. Notice as k increases, our pointers for the source and destination images also increase.

Inside the loop we check to see if blending is on or off. If **blend** is 1, meaning we should blend, we do some fancy math to calculate the color of our blended pixels. The destination value (d) will equal our source value (s) multiplied by our alpha value + our current destination value (d) times 255 minus the **alpha** value. The shift operator (>>8) keeps the value in a 0-255 range.

If blending is disabled (0), we copy the data from the source image directly into the destination image. No blending is done and the **alpha** value is ignored.

```
for( k = 0 ; k < src -> format ; k++, d++, s++)
                                                     // "n" Bytes At A Time
        // Keep in 0-255 Range With >> 8
        else
        *d = *s;
                               // No Blending Just Do A Straight Copy
     }
  d = d + (dst->width - (src_width + dst_xstart))*dst->format; // Add End Of Row
s = s + (src->width - (src_width + src_xstart))*src->format; // Add End Of Row
}
```

The InitGL() code has changed quite a bit. All of the code below is new. We start off by allocating enough memory to hold a 256x256x4 Bytes Per Pixel Image. t1 will point to the allocated ram if everything went well.

After allocating memory for our image, we attempt to load the image. We pass ReadTextureData() the name of the file we wish to open, along with a pointer to our Image Structure (t1).

If we were unable to load the .RAW image, a message box will pop up on the screen to let the user know there was a problem loading the texture.

We then do the same thing for **t2**. We allocate memory, and attempt to read in our second .RAW image. If anything goes wrong we pop up a message box.

```
int InitGL(GLvoid)
                                            // This Will Be Called Right After The GL Wing
{
  t1 = AllocateTextureBuffer( 256, 256, 4 );
                                                          // Get An Image Structure
  if (ReadTextureData("Data/Monitor.raw",t1)==0)
                                                              // Fill The Image Structure
                                // Nothing Read?
   {
     MessageBox(NULL, "Could Not Read 'Monitor.raw' Image Data", "TEXTURE ERROR", MB_OK | MB_
     return FALSE;
  }
  t2 = AllocateTextureBuffer( 256, 256, 4);
                                                         // Second Image Structure
  if (ReadTextureData("Data/GL.raw",t2)==0)
                                                         // Fill The Image Structure With
                                 // Nothing Read?
   {
     MessageBox(NULL, "Could Not Read 'GL.raw' Image Data", "TEXTURE ERROR", MB_OK | MB_ICON:
     return FALSE;
   }
```

If we got this far, it's safe to assume the memory has been allocated and the images have been loaded. Now to use our Blit() command to merge the two images into one.

We start off by passing Blit() **t2** and **t1**, both point to our **TEXTURE_IMAGE** structures (**t2** is the second image, **t1** is the first image.

Then we have to tell blit where to start grabbing data from on the source image. If you load the source image into Adobe Photoshop or any other program capable of loading .RAW images you will see that the entire image is blank except for the top right corner. The top right has a picture of the ball with GL written on it. The bottom left corner of the image is 0,0. The top right of the image is the width of the image-1 (255), the height of the image-1 (255). Knowing that we only want to copy 1/4 of the src image (top right), we tell Blit() to start grabbing from 127,127 (center of our source image).

Next we tell blit how many pixels we want to copy from our source point to the right, and from our source point up. We want to grab a 1/4 chunk of our image. Our image is 256x256 pixels, 1/4 of that is 128x128 pixels. All of the source information is done. Blit() now knows that it should copy from 127 on the x axis to 127+128 (255) on the x axis, and from 127 on the y axis to 127+128 (255) on the y axis.

So Blit() knows what to copy, and where to get the data from, but it doesn't know where to put the data once it's gotten it. We want to draw the ball with GL written on it in the middle our the monitor image. You find the center of the destination image (256x256) which is 128x128 and subtract half the width and height of the source image (128x128) which is 64x64. So (128-64) x (128-64) gives us a starting location of 64,64.

Last thing to do is tell our blitter routine we want to blend the two image (A one means blend, a zero means do not blend), and how much to blend the images. If the last value is 0, we blend the images 0%, meaning anything we copy will replace what was already there. If we use a value of 127, the two images blend together at 50%, and if you use 255, the image you are copying will be completely transparent and will not show up at all.

The pixels are copied from image2 (t2) to image1 (t1). The mixed image will be stored in t1.

// Image To Blend In, Original Image, Src Start X & Y, Src Width & Height, Dst Location Blit(t2,t1,127,127,128,128,64,64,1,127); // Call The Blitter Routine

After we have mixed the two images (t1 and t2) together, we build a texture from the combined

images (t1).

}

After the texture has been created, we can deallocate the memory holding our two **TEXTURE_IMAGE** structures.

The rest of the code is pretty standard. We enable texture mapping, depth testing, etc.

```
BuildTexture (t1);
                                          // Load The Texture Map Into Texture Memory
DeallocateTexture( t1 );
                                             // Clean Up Image Memory Because Texture I:
DeallocateTexture( t2 );
                                             // In GL Texture Memory Now
glEnable(GL_TEXTURE_2D);
                                             // Enable Texture Mapping
glShadeModel(GL_SMOOTH);
                                             // Enables Smooth Color Shading
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
                                                      // This Will Clear The Background
glClearDepth(1.0);
                                         // Enables Clearing Of The Depth Buffer
glEnable(GL_DEPTH_TEST);
                                             // Enables Depth Testing
glDepthFunc(GL_LESS);
                                             // The Type Of Depth Test To Do
return TRUE;
```

I shouldn't even have to explain the code below. We move 5 units into the screen, select our single texture, and draw a texture mapped cube. You should notice that both textures are now combined into one. We don't have to render everything twice to map both textures onto the cube. The blitter code combined the images for us.

```
GLvoid DrawGLScene(GLvoid)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                                                                                   // Clear The Screen And 1
                                                       // Reset The View
   glLoadIdentity();
   glTranslatef(0.0f,0.0f,-5.0f);
   glRotatef(xrot,1.0f,0.0f,0.0f);
   glRotatef(yrot,0.0f,1.0f,0.0f);
   glRotatef(zrot,0.0f,0.0f,1.0f);
   glBindTexture(GL_TEXTURE_2D, texture[0]);
   glBegin(GL_QUADS);
       // Front Face
       glNormal3f( 0.0f, 0.0f, 1.0f);
       glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
       glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
       // Back Face
       glNormal3f( 0.0f, 0.0f, -1.0f);
       glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
       glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
       glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
       // Top Face
       glNormal3f( 0.0f, 1.0f, 0.0f);
       glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
       glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
       glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
       // Bottom Face
       glNormal3f( 0.0f,-1.0f, 0.0f);
       glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
       glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
```

}

```
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Right Face
glNormal3f( 1.0f, 0.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Left Face
glNormal3f(-1.0f, 0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glEnd();
xrot+=0.3f;
yrot+=0.4f;
```

The KillGLWindow() code has a few changes. You'll notice the code to switch from fullscreen mode back to your desktop is now at the top of KillGLWindow(). If the user ran the program in fullscreen mode, the first thing we do when we kill the window is try to switch back to the desktop resolution. If the quick way fails to work, we reset the screen using the information stored in **DMsaved**. This should restore us to our orignal desktop settings.

```
// Properly Kill The Window
GLvoid KillGLWindow(GLvoid)
   if (fullscreen)
                                              // Are We In Fullscreen Mode?
   {
      if (!ChangeDisplaySettings(NULL,CDS_TEST)) {
                                                                // If The Shortcut Doesn't Wo
         ChangeDisplaySettings(NULL,CDS_RESET);
                                                            // Do It Anyway (To Get The Valu
         ChangeDisplaySettings(&DMsaved,CDS_RESET);
                                                            // Change Resolution To The Sav
      }
      else
                                      // Not Fullscreen
      {
         ChangeDisplaySettings(NULL,CDS_RESET);
                                                           // Do Nothing
      ShowCursor(TRUE);
                                             // Show Mouse Pointer
   }
   if (hRC)
                                       // Do We Have A Rendering Context?
   ł
      if (!wglMakeCurrent(NULL,NULL))
                                                         // Are We Able To Release The DC And
         MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR", MB_OK | MB_ICONINI
      }
      if (!wglDeleteContext(hRC))
                                                    // Are We Able To Delete The RC?
      {
         MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN ERROR", MB_OK | MB_I(
      hRC=NULL;
                                        // Set RC To NULL
   }
   if (hDC && !ReleaseDC(hWnd,hDC))
                                                       // Are We Able To Release The DC
   {
      MessageBox(NULL, "Release Device Context Failed.", "SHUTDOWN ERROR", MB_OK | MB_ICONINF(
                                       // Set DC To NULL
      hDC=NULL;
   }
   if (hWnd && !DestroyWindow(hWnd))
                                                       // Are We Able To Destroy The Window'
   {
      MessageBox(NULL, "Could Not Release hWnd.", "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION
                                         // Set hWnd To NULL
      hWnd=NULL;
   }
```

```
if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To Unregister Class
{
    MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORM
    hInstance=NULL; // Set hInstance To NULL
}
```

I've made some changes in CreateGLWindow. The changes will hopefully elimintate alot of the problems people are having when they switch to and from from fullscreen mode. I've included the first part of CreateGLWindow() so you can easily follow through the code.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
   GLuint
              PixelFormat;
                                                  // Holds The Results After Searching For A
   WNDCLASS wc;
                                          // Windows Class Structure
   DWORD
             dwExStyle;
                                              // Window Extended Style
   DWORD
                                             // Window Style
             dwStyle;
   fullscreen=fullscreenflag;
                                                   // Set The Global Fullscreen Flag
                 = GetModuleHandle(NULL);
                                               // Grab An Instance For Our Window
   hInstance
   wc.style = CS_HREDRAW | CS_OWNDC;
                                                                 // Redraw On Size, And Own 1
   wc.lpfnWndProc = (WNDPROC) WndProc;
                                                            // WndProc Handles Messages
  wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
                                                 // No Extra Window Data
  wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Load The Default Icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Load The Arrow Deint
wc.hbrBackground = NULL;
                                                // No Extra Window Data
                                                               // Load The Arrow Pointer
                                   // No Background Required For GL
   wc.lpszMenuName = NULL;
wc.lpszClassName = "OpenGL";
                                                      // We Don't Want A Menu
                                                     // Set The Class Name
```

The big change here is that we now save the current desktop resolution, bit depth, etc. before we switch to fullscreen mode. That way when we exit the program, we can set everything back exactly how it was. The first line below copies the display settings into the **DMsaved** Device Mode structure. Nothing else has changed, just one new line of code.

```
EnumDisplaySettings(NULL, ENUM_CURRENT_SETTINGS, &DMsaved); // Save The Current
if (fullscreen)
                                             // Attempt Fullscreen Mode?
{
  DEVMODE dmScreenSettings;
                                                  // Device Mode
  memset(&dmScreenSettings,0,sizeof(dmScreenSettings));
                                                                      // Makes Sure Memory's
  dmScreenSettings.dmSize=sizeof(dmScreenSettings);
                                                                // Size Of The Devmode Stru
  dmScreenSettings.dmPelsWidth = width; // Selected Screen Width
dmScreenSettings.dmPelsHeight = height; // Selected Screen Height
dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pix
                                                             // Selected Bits Per Pixel
  dmScreenSettings.dmFields=DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;
   // Try To Set Selected Mode And Get Results. NOTE: CDS_FULLSCREEN Gets Rid Of Start
   if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
   {
      // If The Mode Fails, Offer Two Options. Quit Or Use Windowed Mode.
      if (MessageBox(NULL, "The Requested Fullscreen Mode Is Not Supported By\nYour Vide
      {
                                            // Windowed Mode Selected. Fullscreen = FALSE
         fullscreen=FALSE;
      }
      else
      ł
         // Pop Up A Message Box Letting User Know The Program Is Closing.
         MessageBox(NULL,"Program Will Now Close.","ERROR",MB_OK MB_ICONSTOP);
         return FALSE;
                                           // Return FALSE
```

```
}
```

}

WinMain() starts out the same as always. Ask the user if they want fullscreen or not, then start the loop.

```
{
  MSG
                                   // Windows Message Structure
       msg;
  BOOL
       done=FALSE;
                                        // Bool Variable To Exit Loop
  // Ask The User Which Screen Mode They Prefer
  if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreen?", MB_
  {
     fullscreen=FALSE;
                                        // Windowed Mode
  }
  // Create Our OpenGL Window
  if (!CreateGLWindow("Andreas Löffler, Rob Fletcher & NeHe's Blitter & Raw Image Loading
  {
     return 0;
                                    // Quit If Window Was Not Created
  }
  while(!done)
                                      // Loop That Runs While done=FALSE
  {
     if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
                                                       // Is There A Message Waiting?
     {
        if (msg.message==WM_QUIT)
                                             // Have We Received A Quit Message?
        {
                                    // If So done=TRUE
          done=TRUE;
        }
                                  // If Not, Deal With Window Messages
        else
        {
           TranslateMessage(&msg);
                                             // Translate The Message
          DispatchMessage(&msg);
                                             // Dispatch The Message
        }
     }
```

I have made some changes to the code below. If the program is not **active** (minimized) we wait for a message with the command WaitMessage(). Everything stops until the program receives a message (usually maximizing the window). What this means is that the program no longer hogs the processor while it's minimized. Thanks to Jim Strong for the suggestion.

```
if (!active)
                                 // Program Inactive?
{
                                    // Wait For A Message / Do Nothing ( NEW ... Th
  WaitMessage();
}
if (keys[VK_ESCAPE])
                                       // Was Escape Pressed?
{
  done=TRUE:
                               // ESC Signalled A Quit
}
if (keys[VK_F1])
                                   // Is F1 Being Pressed?
  keys[VK_F1]=FALSE;
{
                                    // If So Make Key FALSE
                                    // Kill Our Current Window
  KillGLWindow();
  fullscreen=!fullscreen;
                                          // Toggle Fullscreen / Windowed Mode
  // Recreate Our OpenGL Window
```

}

```
if (!CreateGLWindow("Andreas Löffler, Rob Fletcher & NeHe's Blitter & Raw Image Lo
      {
        return 0;
                                  // Quit If Window Was Not Created
      }
   }
                                        // Draw The Scene
  DrawGLScene();
   SwapBuffers(hDC);
                                        // Swap Buffers (Double Buffering)
}
// Shutdown
KillGLWindow();
                                        // Kill The Window
return (msg.wParam);
                                           // Exit The Program
```

Well, that 's it! Now the doors are open for creating some very cool blending effects for your games, engines or even applications. With texture buffers we used in this tutorial you could do more cool effects like real-time plasma or water. When combining these effects all together you're able to do nearly photo-realistic terrain. If something doesn't work in this tutorial or you have suggestions how to do it better, then please don't hesitate to E-Mail me. Thank you for reading and good luck in creating your own special effects!

Some information about Andreas: I'm an 18 years old pupil who is currently studying to be a software engineer. I've been programming for nearly 10 years now. I've been programming in OpenGL for about 1.5 years.

Andreas Löffler - Original Code / HTML Rob Fletcher - Modified Code (Rewrite) Jeff Molofee (NeHe) - Code Modifications / New HTML

* DOWNLOAD Visual C++ Code For This Lesson.

* DOWNLOAD Irix / GLUT Code For This Lesson. (Conversion by Rob Fletcher)

* DOWNLOAD Mac OS Code For This Lesson. (Conversion by Morgan Aldridge)

Lesson 31

Collision Detection and Physically Based Modeling Tutorial

by

Dimitrios Christopoulos (christop@fhw.gr)

The source code upon which this tutorial is based, is from an older contest entry of mine (at OGLchallenge.dhs.org). The theme was Collision Crazy and my entry (which by the way took the 1st place :)) was called Magic Room. It features collision detection, physically based modeling and effects.

Collision Detection

A difficult subject and to be honest as far as I have seen up until now, there has been no easy solution for it. For every application there is a different way of finding and testing for collisions. Of course there are brute force algorithms which are very general and would work with any kind of objects, but they are expensive.

We are going to investigate algorithms which are very fast, easy to understand and to some extent quite flexible. Furthermore importance must be given on what to do once a collision is detected and how to move the objects, in accordance to the laws of physics. We have a lot stuff to cover. Lets review what we are going to learn:

1) Collision Detection

- Moving Sphere Plane
- Moving Sphere Cylinder
- Moving Sphere Moving Sphere

2) Physically Based Modeling

- Collision Response
- Moving Under Gravity Using Euler Equations

3) Special Effects

- Explosion Modeling Using A Fin-Tree Billboard Method
- Sounds Using The Windows Multimedia Library (Windows Only)

4) Explanation Of The Code

• The Code Is Divided Into 5 Files

Lesson31.cpp)	: Main Code For This Tutorial
Image.cpp,	Image.h	: Code To Load Bitmaps
Tmatrix.cpp,	Tmatrix.h	: Classes To Handle Rotations
Tray.cpp,	Tray.h	: Classes To Handle Ray Operations
Tvector.cpp,	Tvector.h	: Classes To Handle Vector Operations

A lot of handy code! The Vector, Ray and Matrix classes are very useful. I used them until now for personal projects of my own.

1) Collision Detection

For the collision detection we are going to use algorithms which are mostly used in ray tracing. Lets first define a ray.

A ray using vector representation is represented using a vector which denotes the start and a vector (usually normalized) which is the direction in which the ray travels. Essentially a ray starts from the start point and travels in the direction of the direction vector. So our ray equation is:

PointOnRay = Raystart + t * Raydirection

t is a float which takes values from [0, infinity).

With 0 we get the start point and substituting other values we get the corresponding points along the ray.

PointOnRay, Raystart, Raydirection, are 3D Vectors with values (x,y,z). Now we can use this ray representation and calculate the intersections with plane or cylinders.

Ray - Plane Intersection Detection

A plane is represented using its Vector representation as:

Xn dot X = d

Xn, X are vectors and d is a floating point value.

Xn is its normal.

X is a point on its surface.

d is a float representing the distance of the plane along the normal, from the center of the coordinate system.

Essentially a plane represents a half space. So all that we need to define a plane is a 3D point and a normal from that point which is perpendicular to that plane. These two vectors form a plane, ie. if we take for the 3D point the vector (0,0,0) and for the normal (0,1,0) we essentially define a plane across x,z axes. Therefore defining a point and a normal is enough to compute the Vector representation of a plane.

Using the vector equation of the plane the normal is substituted as Xn and the 3D point from which the normal originates is substituted as X. The only value that is missing is d which can easily be computed using a dot product (from the vector equation).

(Note: This Vector representation is equivalent to the widely known parametric form of the plane Ax + By + Cz + D=0 just take the three x,y,z values of the normal as A,B,C and set D=-d).

The two equations we have so far are:

PointOnRay = Raystart + t * Raydirection Xn dot X = d

If a ray intersects the plane at some point then there must be some point on the ray which satisfies the plane equation as follows:

Xn dot PointOnRay = d or (Xn dot Raystart) + t * (Xn dot Raydirection) = d

solving for t:

t = (d - Xn dot Raystart) / (Xn dot Raydirection)

replacing d:

```
t= (Xn dot PointOnRay - Xn dot Raystart) / (Xn dot Raydirection)
```

summing it up:

```
t= (Xn dot (PointOnRay - Raystart)) / (Xn dot Raydirection)
```

t represents the distance from the start until the intersection point along the direction of the ray. Therefore substituting t into the ray equation we can get the collision point. There are a few special cases though. If Xn dot Raydirection = 0 then these two vectors are perpendicular (ray runs parallel to plane) and there will be no collision. If t is negative the collision takes place behind the starting point of the ray along the opposite direction and again there is no intersection.

```
int TestIntersionPlane(const Plane& plane,const TVector& position,const TVector& direction
{
    double DotProduct=direction.dot(plane._Normal); // Dot Product Between Plane Nor
    double 12;
    // Determine If Ray Parallel To Plane
    if ((DotProduct<ZERO)&&(DotProduct>-ZERO))
        return 0;
    l2=(plane._Normal.dot(plane._Position-position))/DotProduct; // Find Distance To Coll:
    if (l2<-ZERO) // Test If Collision Behind Start
        return 0;
    pNormal=plane._Normal;
    lamda=12;
    return 1;
}</pre>
```

The code above calculates and returns the intersection. It returns 1 if there is an intersection otherwise it returns 0. The parameters are the plane, the start and direction of the vector, a double (lamda) where the collision distance is stored if there was any, and the returned normal at the collision point.

Ray - Cylinder Intersection

Computing the intersection between an infinite cylinder and a ray is much more complicated that is why I won't explain it here. There is way too much math involved too easily explain and my goal is primarily to give you tools how to do it without getting into alot of detail (this is not a geometry class). If anyone is interested in the theory behind the intersection code, please look at the Graphic Gems II Book (pp 35, intersection of a with a cylinder). A cylinder is represented as a ray, using a start and direction (here it represents the axis) vector and a radius (radius around the axis of the cylinder). The relevant function is:

int TestIntersionCylinder(const Cylinder& cylinder,const TVector& position,const TVector& (

Returns 1 if an intersection was found and 0 otherwise.

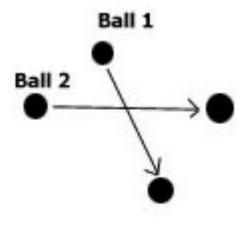
The parameters are the cylinder structure (look at the code explanation further down), the start, direction vectors of the ray. The values returned through the parameters are the distance, the normal at the intersection point and the intersection point itself.

Sphere - Sphere Collision

A sphere is represented using its center and its radius. Determining if two spheres collide is easy. By

finding the distance between the two centers (dist method of the TVector class) we can determine if they intersect, if the distance is less than the sum of their two radius.

The problem lies in determining if 2 MOVING spheres collide. Bellow is an example where 2 sphere move during a time step from one point to another. Their paths cross in-between but this is not enough to prove that an intersection occurred (they could pass at a different time) nor can the collision point be determined.





The previous intersection methods were solving the equations of the objects to determine the intersection. When using complex shapes or when these equations are not available or can not be solved, a different method has to be used. The start points, endpoints, time step, velocity (direction of the sphere + speed) of the sphere and a method of how to compute intersections of static spheres is already known. To compute the intersection, the time step has to be sliced up into smaller pieces. Then we move the spheres according to that sliced time step using its velocity, and check for collisions. If at any point collision is found (which means the spheres have already penetrated each other) then we take the previous position as the intersection point (we could start interpolating between these points to find the exact intersection position, but that is mostly not required).

The smaller the time steps, the more slices we use the more accurate the method is. As an example lets say the time step is 1 and our slices are 3. We would check the two balls for collision at time 0, 0.33, 0.66, 1. Easy !!!!

The code which performs this is:

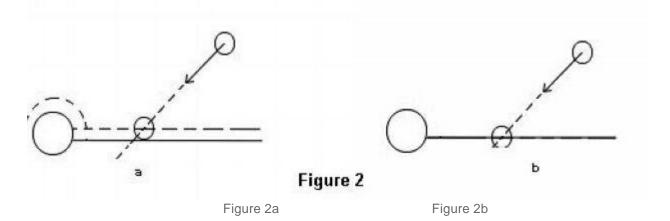
```
/***
                      Find if any of the current balls
                                                                       * * *
/***
                intersect with each other in the current timestep
                                                                       * * *
/*** Returns the index of the 2 intersecting balls, the point and time of intersection ***
int FindBallCol(TVector& point, double& TimePoint, double Time2, int& BallNr1, int& BallNr:
{
  TVector RelativeV;
  TRay rays;
  double MyTime=0.0, Add=Time2/150.0, Timedummy=10000, Timedummy2=-1;
  TVector posi;
                              // Test All Balls Against Eachother In 150 Small Steps
  for (int i=0;i<NrOfBalls-1;i++)</pre>
    for (int j=i+1; j<NrOfBalls; j++)</pre>
       RelativeV=ArrayVel[i]-ArrayVel[j];
                                       // Find Distance
       rays=TRay(OldPos[i],TVector::unit(RelativeV));
       MyTime=0.0;
       if ( (rays.dist(ArrayPos[j])) > 40) continue; // If Distance Between Centers Gre
```

```
// An Intersection Occurred
      while (MyTime<Time2)
                                        // Loop To Find The Exact Intersection Point
         MyTime+=Add;
         posi=OldPos[i]+RelativeV*MyTime;
         if (posi.dist(OldPos[j])<=40)</pre>
            point=posi;
             if (Timedummy>(MyTime-Add)) Timedummy=MyTime-Add;
            BallNr1=i;
            BallNr2=j;
            break;
      }
   }
}
if (Timedummy!=10000)
   TimePoint=Timedummy;
   return 1;
return 0;
```

How To Use What We Just Learned

}

So now that we can determine the intersection point between a ray and a plane/cylinder we have to use it somehow to determine the collision between a sphere and one of these primitives. What we can do so far determine the exact collision point between a particle and a plane/cylinder. The start position of the ray is the position of the particle and the direction of the ray is its velocity (speed and direction). To make it usable fc spheres is quite easy. Look at Figure 2a to see how this can be accomplished.



Each sphere has a radius, take the center of the sphere as the particle and offset the surface along the nc of each plane/cylinder of interest. In Figure 2a these new primitives are represented with dotted lines. You actual primitives of interest are the ones represented by continuous lines, but the collision testing is done to the offset primitives (represented with dotted lines). In essence we perform the intersection test with a little offset plane and a larger in radius cylinder. Using this little trick the ball does not penetrate the surface if a intersection is determined with its center. Otherwise we get a situation as in Figure 2b, where be sphere penetrates the surface. This happens because we determine the intersection between its center and the primitives, which means we did not modify our original code!

Having determined where the collision takes place we have to determine if the intersection takes place in current time step. Timestep is the time we move our sphere from its current point according to its velocity. Because we are testing with infinite rays there is always the possibility that the collision point is after the neposition of the sphere. To determine this we move the sphere, calculate its new position and find the distabetween the start and end point. From our collision detection procedure we also get the distance from the point to its collision point. If this distance is less than the distance between start and end point then there is collision. To calculate the exact time we solve the following simple equation. Represent the distance between

start - end point with Dst, the distance between start - collision point Dsc, and the time step as T. The time where the collision takes place (Tc) is:

Tc= Dsc*T / Dst

All this is performed of course if an intersection is determined. The returned time is a fraction of the whole step, so if the time step was 1 sec, and we found an intersection exactly in the middle of the distance, the calculated collision time would be 0.5 sec. this is interpreted as "0.5 sec after the start there is an intersect Now the intersection point can be calculated by just multiplying Tc with the current velocity and adding it tc start point.

Collision point= Start + Velocity*Tc

This is the collision point on the offset primitive, to find the collision point on the real primitive we add to the point the reverse of the normal at that point (which is also returned by the intersection routines) by the radi the sphere. Note that the cylinder intersection routine returns the intersection point if there is one so it doe need to be calculated.

2) Physically Based Modeling

Collision Response

To determine how to respond after hitting Static Objects like Planes, Cylinders is as important as finding the collision point itself. Using the algorithms and functions described, the exact collision point, the normal at the collision point and the time within a time step in which the collision occurs can be found.

To determine how to respond to a collision, laws of physics have to be applied. When an object collides with the surface its direction changes i.e., it bounces off. The angle of the of the new direction (or reflection vector) with the normal at the collision point is the same as the original direction vector. Figure 3 shows a collision with a sphere.

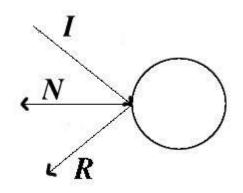


Figure 3

R is the new direction vector I is the old direction vector before the collision N is the Normal at the collision point

The new vector R is calculated as follows:

R= 2*(-I dot N)*N + I

The restriction is that the I and N vectors have to be unit vectors. The velocity vector as used in our

examples represents speed and direction. Therefore it can not be plugged into the equation in the place of I, without any transformation. The speed has to be extracted. The speed for such a velocity vector is extracted finding the magnitude of the vector. Once the magnitude is found, the vector can be transformed to a unit vector and plugged into the equation giving the reflection vector R. R shows us now the direction, of the reflected ray, but in order to be used as a velocity vector it must also incorporate the speed. Therefore it gets, multiplied with the magnitude of the original ray, thus resulting in the correct velocity vector.

In the example this procedure is applied to compute the collision response if a ball hits a plane or a cylinder. But it works also for arbitrary surfaces, it does not matter what the shape of the surface is. As long as a collision point and a Normal can be found the collision response method is always the same. The code which does these operations is:

rt2=ArrayVel[BallNr].mag(); ArrayVel[BallNr].unit(); // Find Magnitude Of Velocity
// Normalize It

// Compute Reflection
ArrayVel[BallNr]=TVector::unit((normal*(2*normal.dot(-ArrayVel[BallNr]))) + ArrayVel[BallN
ArrayVel[BallNr]=ArrayVel[BallNr]*rt2; // Muliply With Magnitude To Obtain F:

When Spheres Hit Other Spheres Determining the collision response, if two balls hit each other is much more difficult. Complex equations of particle dynamics have to be solved and therefore I will just post the final solution without any proof. Just trust me on this one :) During the collision of 2 balls we have a situation as it is depicted in Figure 4.

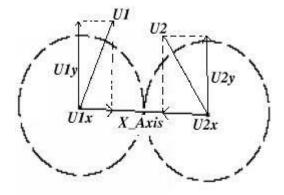


Figure 4

U1 and U2 are the velocity vectors of the two spheres at the time of impact. There is an axis (X_Axis) vector which joins the 2 centers of the spheres, and U1x, U2x are the projected vectors of the velocity vectors U1,U2 onto the axis (X_Axis) vector.

U1y and U2y are the projected vectors of the velocity vectors U1,U2 onto the axis which is perpendicular to the X_Axis. To find these vectors a few simple dot products are needed. M1, M2 is the mass of the two spheres respectively. V1,V2 are the new velocities after the impact, and V1x, V1y, V2x, V2y are the projections of the velocity vectors onto the X_Axis.

In More Detail:

a) Find X_Axis

X_Axis = (center2 - center1); Unify X_Axis, X_Axis.unit();

b) Find Projections

U1x= X_Axis * (X_Axis dot U1) U1y= U1 - U1x U2x =-X_Axis * (-X_Axis dot U2) U2y =U2 - U2x

c)Find New Velocities

(U1x * M1)+(U2x*M2)-(U1x-U2x)*M2 V1x= -----M1+M2 (U1x * M1)+(U2x*M2)-(U2x-U1x)*M1 V2x= -----M1+M2

In our application we set the M1=M2=1, so the equations get even simpler.

d)Find The Final Velocities

V1y=U1y V2y=U2y V1=V1x+V1y V2=V2x+V2y

The derivation of that equations has a lot of work, but once they are in a form like the above they can be used quite easily. The code which does the actual collision response is:

```
TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
double a.b;
pb1=OldPos[BallColNr1]+ArrayVel[BallColNr1]*BallTime;// Find Position Of Ball1pb2=OldPos[BallColNr2]+ArrayVel[BallColNr2]*BallTime;// Find Position Of Ball2
                                   // Find X-Axis
xaxis=(pb2-pb1).unit();
a=xaxis.dot(ArrayVel[BallColNr1]);
                                               // Find Projection
                                  // Find Projected Vectors
Ulx=xaxis*a;
Uly=ArrayVel[BallColNr1]-Ulx;
xaxis=(pb1-pb2).unit();
                                          // Do The Same As Above
// To Find Projection
U2y=ArrayVel[BallColNr2]-U2x;
V1x=(U1x+U2x-(U1x-U2x))*0.5;
                                            // Now Find New Velocities
V2x=(U1x+U2x-(U2x-U1x))*0.5;
Vly=Uly;
V2y=U2y;
for (j=0;j<NrOfBalls;j++)</pre>
                                         // Update All Ball Positions
ArrayPos[j]=OldPos[j]+ArrayVel[j]*BallTime;
ArrayVel[BallColNr1]=V1x+V1y;
                                            // Set New Velocity Vectors
ArrayVel[BallColNr2]=V2x+V2y;
                                            // To The Colliding Balls
```

Moving Under Gravity Using Euler Equations

To simulate realistic movement with collisions, determining the the collision point and computing the response is not enough. Movement based upon physical laws also has to be simulated.

The most widely used method for doing this is using Euler equations. As indicated all the computations are going to be performed using time steps. This means that the whole simulation is advanced in certain time steps during which all the movement, collision and response tests are performed. As an example we can advanced a simulation 2 sec. on each frame. Based on Euler equations, the velocity and position at each new time step is computed as follows:

```
Velocity_New = Velovity_Old + Acceleration*TimeStep
Position_New = Position_Old + Velocity_New*TimeStep
```

Now the objects are moved and tested angainst collision using this new velocity. The Acceleration

for each object is determined by accumulating the forces which are acted upon it and divide by its mass according to this equation:

Force = mass * acceleration

A lot of physics formulas :)

But in our case the only force the objects get is the gravity, which can be represented right away as a vector indicating acceleration. In our case something negative in the Y direction like (0,-0.5,0). This means that at the beginning of each time step, we calculate the new velocity of each sphere and move them testing for collisions. If a collision occurs during a time step (say after 0.5 sec with a time step equal to 1 sec.) we advance the object to this position, compute the reflection (new velocity vector) and move the object for the remaining time (which is 0.5 in our example) testing again for collisions during this time. This procedure gets repeated until the time step is completed.

When multiple moving objects are present, each moving object is tested with the static geometry for intersections and the nearest intersection is recorded. Then the intersection test is performed for collisions among moving objects, where each object is tested with everyone else. The returned intersection is compared with the intersection returned by the static objects and the closest one is taken. The whole simulation is updated to that point, (i.e. if the closest intersection would be after 0.5 sec. we would move all the objects for 0.5 seconds), the reflection vector is calculated for the colliding object and the loop is run again for the remaining time.

3) Special Effects

Explosions

Every time a collision takes place an explosion is triggered at the collision point. A nice way to model explosions is to alpha blend two polygons which are perpendicular to each other and have as the center the point of interest (here intersection point). The polygons are scaled and disappear over time. The disappearing is done by changing the alpha values of the vertices from 1 to 0, over time. Because a lot of alpha blended polygons can cause problems and overlap each other (as it is stated in the Red Book in the chapter about transparency and blending) because of the Z buffer, we borrow a technique used in particle rendering. To be correct we had to sort the polygons from back to front according to their eye point distance, but disabling the Depth buffer writes (not reads) also does the trick (this is also documented in the red book). Notice that we limit our number of explosions to maximum 20 per frame, if additional explosions occur and the buffer is full, the explosion is discarded. The source which updates and renders the explosions is:

```
// Render / Blend Explosions
glEnable(GL_BLEND);
                                        // Enable Blending
                                           // Disable Depth Buffer Writes
glDepthMask(GL_FALSE);
glBindTexture(GL_TEXTURE_2D, texture[1]);
                                               // Upload Texture
for(i=0; i<20; i++)</pre>
                                        // Update And Render Explosions
ł
   if(ExplosionArray[i]._Alpha>=0)
   {
      glPushMatrix();
      ExplosionArray[i]._Alpha-=0.01f;
                                               // Update Alpha
      ExplosionArray[i]._Scale+=0.03f;
                                               // Update Scale
      // Assign Vertices Colour Yellow With Alpha
      // Colour Tracks Ambient And Diffuse
      glColor4f(1,1,0,ExplosionArray[i]._Alpha);
                                                     // Scale
      glScalef(ExplosionArray[i]._Scale,ExplosionArray[i]._Scale,ExplosionArray[i]._Scale)
      // Translate Into Position Taking Into Account The Offset Caused By The Scale
      glTranslatef((float)ExplosionArray[i]._Position.X()/ExplosionArray[i]._Scale,(float)]
         (float)ExplosionArray[i]._Position.Z()/ExplosionArray[i]._Scale);
      glCallList(dlist);
                                       // Call Display List
      glPopMatrix();
   }
}
```

Sound

For the sound the windows multimedia function PlaySound() is used. This is a quick and dirty way to play wav files quickly and without trouble.

4) Explaining the Code

Congratulations...

If you are still with me you have survived successfully the theory section ;) Before having fun playing around with the demo, some further explanations about the source code are necessary. The main flow and steps of the simulation are as follows (in pseudo code):

```
While (Timestep!=0)
   {
      For each ball
        compute nearest collision with planes;
        compute nearest collision with cylinders;
        Save and replace if it the nearest intersection
        in time computed until now;
      Check for collision among moving balls;
      Save and replace if it the nearest intersection
      in time computed until now;
      If (Collision occurred)
      {
         Move All Balls for time equal to collision time;
         (We already have computed the point, normal and collision time.)
         Compute Response;
         Timestep-=CollisonTime;
       }
       else
         Move All Balls for time equal to Timestep
    }
```

The actual code which implements the above pseudo code is much harder to read but essentially is an exact implementation of the pseudo code above.

```
// While Time Step Not Over
while (RestTime>ZERO)
  lamda=10000;
                                    // Initialize To Very Large Value
   // For All The Balls Find Closest Intersection Between Balls And Planes / Cylinders
  for (int i=0;i<NrOfBalls;i++)</pre>
      // Compute New Position And Distance
     OldPos[i]=ArrayPos[i];
     TVector::unit(ArrayVel[i],uveloc);
     ArrayPos[i]=ArrayPos[i]+ArrayVel[i]*RestTime;
     rt2=OldPos[i].dist(ArrayPos[i]);
      // Test If Collision Occured Between Ball And All 5 Planes
      if (TestIntersionPlane(pl1,OldPos[i],uveloc,rt,norm))
         // Find Intersection Time
        rt4=rt*RestTime/rt2;
         // If Smaller Than The One Already Stored Replace In Timestep
         if (rt4<=lamda)
            // If Intersection Time In Current Time Step
```

```
if (rt4<=RestTime+ZERO)
            if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
            {
               normal=norm;
               point=OldPos[i]+uveloc*rt;
               lamda=rt4;
               BallNr=i;
            }
      }
   }
   if (TestIntersionPlane(pl2,OldPos[i],uveloc,rt,norm))
      // ... The Same As Above Omitted For Space Reasons
   }
   if (TestIntersionPlane(pl3,OldPos[i],uveloc,rt,norm))
   {
      // ... The Same As Above Omitted For Space Reasons
   if (TestIntersionPlane(pl4,OldPos[i],uveloc,rt,norm))
      // ... The Same As Above Omitted For Space Reasons
   }
   if (TestIntersionPlane(pl5,OldPos[i],uveloc,rt,norm))
   {
      // ... The Same As Above Omitted For Space Reasons
   }
   // Now Test Intersection With The 3 Cylinders
   if (TestIntersionCylinder(cyl1,OldPos[i],uveloc,rt,norm,Nc))
   Ł
      rt4=rt*RestTime/rt2;
      if (rt4<=lamda)
      ł
         if (rt4<=RestTime+ZERO)
            if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
            {
               normal=norm;
               point=Nc;
               lamda=rt4;
               BallNr=i;
            }
      }
   }
   if (TestIntersionCylinder(cyl2,OldPos[i],uveloc,rt,norm,Nc))
   ł
      // ... The Same As Above Omitted For Space Reasons
   }
   if (TestIntersionCylinder(cyl3,OldPos[i],uveloc,rt,norm,Nc))
   {
      // ... The Same As Above Omitted For Space Reasons
   }
}
// After All Balls Were Tested With Planes / Cylinders Test For Collision
// Between Them And Replace If Collision Time Smaller
if (FindBallCol(Pos2,BallTime,RestTime,BallColNr1,BallColNr2))
   if (sounds)
      PlaySound("Explode.wav",NULL,SND_FILENAME|SND_ASYNC);
```

```
if ( (lamda==10000) || (lamda>BallTime) )
   {
      RestTime=RestTime-BallTime;
      TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
      double a,b;
      .
      Code Omitted For Space Reasons
      The Code Is Described In The Physically Based Modeling
      Section Under Sphere To Sphere Collision
      //Update Explosion Array And Insert Explosion
      for(j=0;j<20;j++)</pre>
      {
         if (ExplosionArray[j]._Alpha<=0)</pre>
         {
            ExplosionArray[j]._Alpha=1;
            ExplosionArray[j]._Position=ArrayPos[BallColNr1];
            ExplosionArray[j]._Scale=1;
            break;
      }
      continue;
   }
}
// End Of Tests
// If Collision Occured Move Simulation For The Correct Timestep
// And Compute Response For The Colliding Ball
if (lamda!=10000)
{
   RestTime-=lamda;
   for (j=0;j<NrOfBalls;j++)</pre>
   ArrayPos[j]=OldPos[j]+ArrayVel[j]*lamda;
   rt2=ArrayVel[BallNr].mag();
   ArrayVel[BallNr].unit();
   ArrayVel[BallNr]=TVector::unit( (normal*(2*normal.dot(-ArrayVel[BallNr]))) + ArrayVe
   ArrayVel[BallNr]=ArrayVel[BallNr]*rt2;
   // Update Explosion Array And Insert Explosion
   for(j=0;j<20;j++)</pre>
   {
      if (ExplosionArray[j]._Alpha<=0)</pre>
      {
         ExplosionArray[j]._Alpha=1;
         ExplosionArray[j]._Position=point;
         ExplosionArray[j]._Scale=1;
         break;
      }
   }
else RestTime=0;
                          // End Of While Loop
```

The Main Global Variables Of Importance Are:

}

Represent the direction and position of the camera. The camera is moved using the LookAt function. As you will probably notice, if not in hook mode (which I will explain later), the whole scene rotates around, the degree of rotation is handled with camera_rotation.	TVector dir TVector pos(0,- 50,1000); float camera_rotation=0;
Represent the acceleration applied to the moving balls. Acts as gravity in the application.	TVector accel(0,- 0.05,0);
Arrays which hold the New and old ball positions and the velocity vector of	TVector ArrayVel[10]; TVector ArrayPos

	[10]; TVector OldPos[10]; int NrOfBalls=3;
The time step we use.	double Time=0.6;
If 1 the camera view changes and a (the ball with index 0 in the array) ball is followed. For making the camera following the ball we used its position and velocity vector to position the camera exactly behind the ball and make it look along the velocity vector of the ball.	int hook_toball1=0;
ISalt evolgengtory structures for holding data about evolgeions, planes and	struct Plane struct Cylinder struct Explosion
The explosions are stored in a array, of fixed length.	Explosion ExplosionArray[20];

The Main Functions Of Interest Are:

	int TestIntersionPlane(); int TestIntersionCylinder ();
Loads Textures from bmp files	void LoadGLTextures();
Has the rendering code. Renders the balls, walls, columns and explosions	void DrawGLScene();
Performs the main simulation logic	void idle();
Sets Up OpenGL state	void InitGL();
Find if any balls collide again each other in current time step	int FindBallCol();

For more information look at the source code. I tried to comment it as best as I could. Once the collision detection and response logic is understood, the source should become very clear. For any more info don't hesitate to contact me.

As I stated at the beginning of this tutorial, the subject of collision detection is a very difficult subject to cover in one tutorial. You will learn a lot in this tutorial, enough to create some pretty impressive demos of your own, but there is still alot more to learn on this subject. Now that you have the basics, all the other sources on Collision Detection and Physically Based Modeling out there should become easier to understand. With this said, I send you on your way and wish you happy collisions!!!

Some information about Dimitrios Christopoulos: He is currently working as a Virtual Reality software engineer at the Foundation of the Hellenic World in Athens/Greece (www.fhw.gr). Although Born in Germany, he studied in Greece at the University of Patras for a B.Sc. in Computer Engineering and Informatics. He holds also a MSc degree (honours) from the University of Hull (UK) in Computer Graphics and Virtual Environments. He did his first steps in game programming using Basic on an Commodore 64, and switched to C/C++/Assembly on the PC platform after the start of his studium. During the last few years OpenGL has become his graphics API of choice. For more information visit his Homepage.

Dimitrios Christopoulos - Code / HTML Jeff Molofee (NeHe) - HTML Modifications

- * DOWNLOAD Visual C++ Code For This Lesson.
- * DOWNLOAD Mac OS Code For This Lesson. (Conversion by Morgan Aldridge)
- * DOWNLOAD Borland C++ Builder 4.0 Code For This Lesson. (Conversion by Dave Rowbotham)

Lesson 32

Model Rendering Tutorial

by

Brett Porter (brettporter@yahoo.com)

The source for this project has been extracted from PortaLib3D, a library I have written to enable users to do things like displaying models with very little extra code. But so that you can trust such a library, you should understand what it is doing, so this tutorial aims to help with that.

The portions of PortaLib3D included here retain my copyright notices. This doesn't mean they can't be used by you - it means that if you cut-and-paste the code into your project, you have to give me proper credit. That's all. If you choose to read, understand, and re-implement the code yourself (and it is what you are encouraged to do if you are not actually using the library. You don't learn anything with cut-and-paste!), then you free yourself of that obligation. Let's face it, the code is nothing special. Ok, let's get onto something more interesting!

OpenGL Base Code

The OpenGL base code is in **Lesson32.cpp**. Mostly it came from Lesson 6, with a small modification to the loading of textures and the drawing routine. The changes will be discussed later.

Milkshape 3D

The model I use in this example is from Milkshape 3D. The reason I use this is because it is a damn fine modelling package, and it includes its file-format so it is easy to parse and understand. My next plan is to implement an Anim8or file reader because it is free and of course a 3DS reader.

However, the file format, while it will be described briefly here, is not the major concern for loading a model. You must create your own structures that are suitable to store the data, and then read the file into that. So first, let's describe the structures required for a model.

Model Data Structures

These model data structures come from the class Model in **Model.h**. First, and most important, we need vertices:

```
// Vertex Structure
struct Vertex
{
    char m_boneID; // For Skeletal Animation
    float m_location[3];
};
// Vertices Used
int m_numVertices;
Vertex *m_pVertices;
```

For now, you can ignore the m_boneID variable - that will come in a future tutorial! The m_location

array represents the coordinate of the vertex (X,Y,Z). The two variables store the number of vertices and the actual vertices in a dynamic array which is allocated by the loader.

Next we need to group these vertices into triangles:

```
// Triangle Structure
struct Triangle
{
    float m_vertexNormals[3][3];
    float m_s[3], m_t[3];
    int m_vertexIndices[3];
};
// Triangles Used
int m_numTriangles;
Triangle *m_pTriangles;
```

Now, the 3 vertices that make up the triangle are stored in m_vertexIndices. These are offsets into the array of m_pVertices. This way each vertex need only be listed once, saving memory (and calculations when it comes to animating later). m_s and m_t are the (s,t) texture coordinates for each of the 3 vertices. The texture used is the one applied to this mesh (which is described next). Finally we have the m_vertexNormals member which stores the normal to each of the 3 vertices. Each normal has 3 float coordinates describing the vector.

The next structure we have in a model is a mesh. A mesh is a group of triangles that all have the same material applied to them. The collection of meshes make up the entire model. The mesh structure is as follows:

```
// Mesh
struct Mesh
{
    int m_materialIndex;
    int m_numTriangles;
    int *m_pTriangleIndices;
};
// Meshes Used
```

int m_numMeshes; Mesh *m_pMeshes;

This time you have m_pTriangleIndices storing the triangles in the mesh in the same way as the triangle stored indicies to its vertices. It will be dynamically allocated because the number of triangles in a mesh is not known in advance, and is specified by m_numTriangles. Finally, m_materialIndex is the index of the material (texture and lighting coeffecients) to use for the mesh. I'll show you the material structure below:

```
// Material Properties
struct Material
{
    float m_ambient[4], m_diffuse[4], m_specular[4], m_emissive[4];
    float m_shininess;
    GLuint m_texture;
    char *m_pTextureFilename;
};
// Materials Used
int m_numMaterials;
Material *m_pMaterials;
```

Here we have all the standard lighting coeffecients in the same format as OpenGL: ambient, diffuse, specular, emissive and shininess. We also have the texture object m_texture and the filename (dynamically allocated) of the texture so that it can be reloaded if the OpenGL context is lost.

The Code - Loading the Model

Now, on to loading the model. You will notice there is a pure virtual function called loadModelData, which takes the filename of the model as an argument. What happens is we create a derived class, MilkshapeModel, which implements this function, filling in the protected data structures mentioned above. Lets look at that function now:

```
bool MilkshapeModel::loadModelData( const char *filename )
{
    ifstream inputFile( filename, ios::in | ios::binary | ios::nocreate );
    if ( inputFile.fail())
        return false; // "Couldn't Open The Model File."
```

First, the file is opened. It is a binary file, hence the ios::binary qualifier. If it is not found, the function returns false to indicate an error.

```
inputFile.seekg( 0, ios::end );
long fileSize = inputFile.tellg();
inputFile.seekg( 0, ios::beg );
```

The above code determines the size of the file in bytes.

```
byte *pBuffer = new byte[fileSize];
inputFile.read( pBuffer, fileSize );
inputFile.close();
```

Then the file is read into a temporary buffer in its entirety.

```
const byte *pPtr = pBuffer;
MS3DHeader *pHeader = ( MS3DHeader* )pPtr;
pPtr += sizeof( MS3DHeader );
if ( strncmp( pHeader->m_ID, "MS3D000000", 10 ) != 0 )
  return false; // "Not A Valid Milkshape3D Model File."
if ( pHeader->m_version < 3 || pHeader->m_version > 4 )
  return false; // "Unhandled File Version. Only Milkshape3D Version 1.3 And 1.4 Is
```

Now, a pointer is acquired to out current position in the file, pPtr. A pointer to the header is saved, and then the pointer is advanced past the header. You will notice several MS3D... structures being used here. These are declared at the top of **MilkshapeModel.cpp**, and come directly from the file format specification. The fields of the header are checked to make sure that this is a valid file we are reading.

```
int nVertices = *( word* )pPtr;
m_numVertices = nVertices;
m_pVertices = new Vertex[nVertices];
pPtr += sizeof( word );
int i;
for ( i = 0; i < nVertices; i++ )
{
    MS3DVertex *pVertex = ( MS3DVertex* )pPtr;
    m_pVertices[i].m_boneID = pVertex->m_boneID;
    memcpy( m_pVertices[i].m_location, pVertex->m_vertex, sizeof( float )*3 );
    pPtr += sizeof( MS3DVertex );
}
```

The above code reads each of the vertex structures in the file. First memory is allocated in the model for the vertices, and then each is parsed from the file as the pointer is advanced. Several calls to memory will be used in this function, which copies the contents of the small arrays easily. The m_boneID member can still be ignored for now - its for skeletal animation!

```
int nTriangles = *( word* )pPtr;
m_numTriangles = nTriangles;
m_pTriangles = new Triangle[nTriangles];
pPtr += sizeof( word );
for ( i = 0; i < nTriangles; i++ )
{
    MS3DTriangle *pTriangle = ( MS3DTriangle* )pPtr;
    int vertexIndices[3] = { pTriangle->m_vertexIndices[0], pTriangle->m_vertexIndices[1]
    float t[3] = { 1.0f-pTriangle->m_t[0], 1.0f-pTriangle->m_t[1], 1.0f-pTriangle->m_t[2]
    memcpy( m_pTriangles[i].m_vertexNormals, pTriangle->m_vertexNormals, sizeof( float )*
    memcpy( m_pTriangles[i].m_s, pTriangle->m_s, sizeof( float )*3 );
    memcpy( m_pTriangles[i].m_t, t, sizeof( float )*3 );
    memcpy( m_pTriangles[i].m_vertexIndices, vertexIndices, sizeof( int )*3 );
    pPtr += sizeof( MS3DTriangle );
}
```

As for the vertices, this part of the function stores all of the triangles in the model. While most of it involves just copying the arrays from one structure to another, you'll notice the difference for the vertexIndices and t arrays. In the file, the vertex indices are stores as an array of word values, but in the model they are int values for consistency and simplicity (no nasty casting needed). So this just converts the 3 values to integers. The t values are all set to 1.0-(original value). The reason for this is that OpenGL uses a lower-left coordinate system, whereas Milkshape uses an upper-left coordinate system for its texture coordinates.

```
int nGroups = *( word* )pPtr;
m_numMeshes = nGroups;
m_pMeshes = new Mesh[nGroups];
pPtr += sizeof( word );
for ( i = 0; i < nGroups; i++ )
{
  pPtr += sizeof( byte );
                           // Flags
  pPtr += 32;
                   // Name
  word nTriangles = *( word* )pPtr;
  pPtr += sizeof( word );
   int *pTriangleIndices = new int[nTriangles];
   for ( int j = 0; j < nTriangles; j++ )
     pTriangleIndices[j] = *( word* )pPtr;
      pPtr += sizeof( word );
   char materialIndex = *( char* )pPtr;
```

```
pPtr += sizeof( char );
m_pMeshes[i].m_materialIndex = materialIndex;
m_pMeshes[i].m_numTriangles = nTriangles;
m_pMeshes[i].m_pTriangleIndices = pTriangleIndices;
}
```

The above code loads the mesh data structures (also called groups in Milkshape3D). Since the number of triangles varies from mesh to mesh, there is no standard structure to read. Instead, they are taken field by field. The memory for the triangle indices is dynamically allocated within the mesh and read one at a time.

```
int nMaterials = *( word* )pPtr;
m_numMaterials = nMaterials;
m_pMaterials = new Material[nMaterials];
pPtr += sizeof( word );
for ( i = 0; i < nMaterials; i++ )
{
  MS3DMaterial *pMaterial = ( MS3DMaterial* )pPtr;
  memcpy( m_pMaterials[i].m_ambient, pMaterial->m_ambient, sizeof( float )*4 );
  memcpy( m_pMaterials[i].m_diffuse, pMaterial->m_diffuse, sizeof( float )*4 );
  memcpy( m_pMaterials[i].m_specular, pMaterial->m_specular, sizeof( float )*4 );
  memcpy( m_pMaterials[i].m_emissive, pMaterial->m_emissive, sizeof( float )*4 );
  m_pMaterials[i].m_shininess = pMaterial->m_shininess;
  m_pMaterials[i].m_pTextureFilename = new char[strlen( pMaterial->m_texture )+1];
  strcpy( m_pMaterials[i].m_pTextureFilename, pMaterial->m_texture );
  pPtr += sizeof( MS3DMaterial );
}
```

```
reloadTextures();
```

Lastly, the material information is taken from the buffer. This is done in the same way as those above, copying each of the lighting coefficients into the new structure. Also, new memory is allocated for the texture filename, and it is copied into there. The final call to reloadTextures is used to actually load the textures and bind them to OpenGL texture objects. That function, from the Model base class, is described later.

```
delete[] pBuffer;
```

```
return true;
```

}

The last fragment frees the temporary buffer now that all the data has been copied and returns successfully.

So at this point, the protected member variables of the Model class are filled with the model information. You'll note also that this is the only code in MilkshapeModel because it is the only code specific to Milkshape3D. Now, before the model can be rendered, it is necessary to load the textures for each of its materials. This is done with the following code:

```
void Model::reloadTextures()
{
    for ( int i = 0; i < m_numMaterials; i++ )
        if ( strlen( m_pMaterials[i].m_pTextureFilename ) > 0 )
            m_pMaterials[i].m_texture = LoadGLTexture( m_pMaterials[i].m_pTextureFilename );
        else
            m_pMaterials[i].m_texture = 0;
}
```

For each material, the texture is loaded using a function from NeHe's base code (slightly modified from it's previous version). If the texture filename was an empty string, then it is not loaded, and instead the texture object identifier is set to 0 to indicate there is no texture.

The Code - Drawing the Model

Now we can start the code to draw the model! This is not difficult at all now that we have a careful arrangement of the data structures in memory.

```
void Model::draw()
{
    GLboolean texEnabled = gllsEnabled( GL_TEXTURE_2D );
```

This first part saves the state of texture mapping within OpenGL so that the function does not disturb it. Note however that it does not preserve the material properties in the same way.

Now we loop through each of the meshes and draw them individually:

```
// Draw By Group
for ( int i = 0; i < m_numMeshes; i++ )
{</pre>
```

m_pMeshes[i] will be used to reference the current mesh. Now, each mesh has its own material properties, so we set up the OpenGL states according to that. If the materialIndex of the mesh is -1 however, there is no material for this mesh and it is drawn with the OpenGL defaults.

```
int materialIndex = m_pMeshes[i].m_materialIndex;
if ( materialIndex >= 0 )
{
  glMaterialfv( GL_FRONT, GL_AMBIENT, m_pMaterials[materialIndex].m_ambient );
  glMaterialfv( GL_FRONT, GL_DIFFUSE, m_pMaterials[materialIndex].m_diffuse );
  glMaterialfv( GL_FRONT, GL_SPECULAR, m_pMaterials[materialIndex].m_specular );
  glMaterialfv( GL_FRONT, GL_EMISSION, m_pMaterials[materialIndex].m_emissive );
  glMaterialf( GL_FRONT, GL_SHININESS, m_pMaterials[materialIndex].m_shininess );
  if ( m_pMaterials[materialIndex].m_texture > 0 )
   ł
      glBindTexture( GL_TEXTURE_2D, m_pMaterials[materialIndex].m_texture );
     glEnable( GL_TEXTURE_2D );
  else
     glDisable( GL_TEXTURE_2D );
}
else
{
  glDisable( GL_TEXTURE_2D );
}
```

The material properties are set according to the values stored in the model. Note that the texture is only bound and enabled if it is greater than 0. If it is set to 0, you'll recall, there was no texture, so texturing is disabled. Texturing is also disabled if there was no material at all for the mesh.

```
glBegin( GL_TRIANGLES );
{
```

```
for ( int j = 0; j < m_pMeshes[i].m_numTriangles; j++ )
{
    int triangleIndex = m_pMeshes[i].m_pTriangleIndices[j];
    const Triangle* pTri = &m_pTriangles[triangleIndex];
    for ( int k = 0; k < 3; k++ )
    {
        int index = pTri->m_vertexIndices[k];
        glNormal3fv( pTri->m_vertexNormals[k] );
        glTexCoord2f( pTri->m_s[k], pTri->m_t[k] );
        glVertex3fv( m_pVertices[index].m_location );
    }
    }
    glEnd();
```

The above section does the rendering of the triangles for the model. It loops through each of the triangles for the mesh, and then draws each of its three vertices, including the normal and texture coordinates. Remember that each triangle in a mesh and likewise each vertex in a triangle is indexed into the total model arrays (these are the two index variables used). pTri is a pointer to the current triangle in the mesh used to simplify the code following it.

```
if ( texEnabled )
   glEnable( GL_TEXTURE_2D );
else
   glDisable( GL_TEXTURE_2D );
```

}

}

This final fragment of code sets the texture mapping state back to its original value.

The only other code of interest in the Model class is the constructor and destructor. These are self explanatory. The constructor initializes all members to 0 (or NULL for pointers), and the destructor deletes the dynamic memory for all of the model structures. You should note that if you call the loadModelData function twice for one Model object, you will get memory leaks. Be careful!

The final topic I will discuss here is the changes to the base code to render using the new Model class, and where I plan to go from here in a future tutorial introducing skeletal animation.

Model *pModel = NULL; // Holds The Model Data

At the top of the code in **Lesson32.cpp** the model is declared, but not initialised. It is created in WinMain:

The model is created here, and **not** in InitGL because InitGL gets called everytime we change the screen mode (losing the OpenGL context). But the model doesn't need to be reloaded, as its data remains intact. What doesn't remain intact are the textures that were bound to texture objects when we loaded the object. So the following line is added to InitGL:

pModel->reloadTextures();

This takes the place of calling LoadGLTextures as we used to. If there was more than one model in the scene, then this function must be called for all of them. If you get white objects all of a sudden, then your textures have been thrown away and not reloaded correctly.

Finally there is a new DrawGLScene function:

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth
    glLoadIdentity(); // Reset The View
    gluLookAt( 75, 75, 75, 0, 0, 0, 0, 1, 0);
    glRotatef(yrot,0.0f,1.0f,0.0f);
    pModel->draw();
    yrot+=1.0f;
    return TRUE; // Keep Going
}
```

Simple? We clear the colour buffer, set the identity into the model/view matrix, and then set an eye projection with gluLookAt. If you haven't used gluLookAt before, essentially it places the camera at the position of the first 3 parameters, places the center of the scene at the position of the next 3 parameters, and the last 3 parameters describe the vector that is "up". In this case, we look from (75, 75, 75) to (0,0,0) - as the model is drawn about (0,0,0) unless you translate before drawing it - and the positive Y-axis is facing up. The function must be called first, and after loading the identity to behave in this fashion.

To make it a bit more interesting, the scene gradually rotates around the y-axis with glRotatef.

Finally, the model is drawn with its draw member function. It is drawn centered at the origin (assuming it was modelled around the origin in Milkshape 3D!), so If you want to position or rotate or scale it, simply call the appropriate GL functions before drawing it. Voila! To test it out - try making your own models in Milkshape (or use its import function), and load them instead by changing the line in WinMain. Or add them to the scene and draw several models!

What Next?

In a future tutorial for NeHe Productions, I will explain how to extend this class structure to incorporate skeletal animation. And if I get around to it, I will write more loader classes to make the program more versatile.

The step to skeletal animation is not as large as it may seem, although the math involved is much more tricky. If you don't understand much about matrices and vectors, now is the time to read up them! There are several resources on the web that can help you out.

See you then!

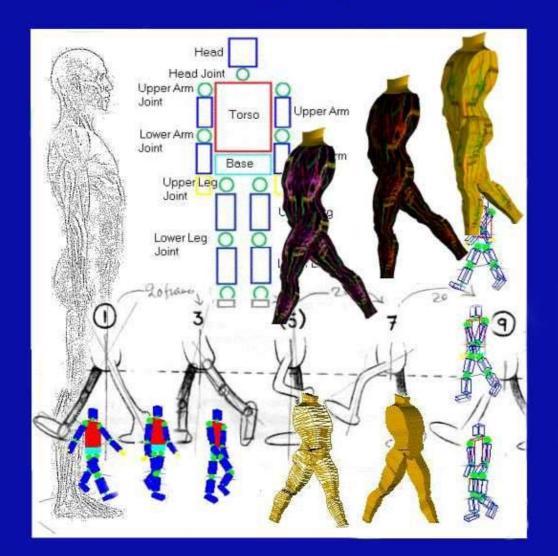
Some information about Brett Porter: He is currently working as a Java programmer for IDM's gameplayNOW. Born in Australia, he studied at the University of Wollongong, recently graduating with a BCompSc and a BMath. He began programming in BASIC 12 years ago on a Commodore 64 "clone" called the VZ300, but soon moved up to Pascal, Intel assembly, C++ and Java. During the last few years 3D programming has become an interest and OpenGL has become his graphics API of choice. For more information visit his Homepage.

Brett Porter - Code / HTML Jeff Molofee (NeHe) - HTML Modifications * DOWNLOAD Visual C++ Code For This Lesson.

* DOWNLOAD Borland C++ Builder 4.0 Code For This Lesson. (Conversion by Dave Rowbotham)

The Developers Gallery

www.dev-gallery.com



A 3D Case Study Using OpenGL

Table of Contents

Disclaimer	<u>.</u>	1
-	<u>– Introduction</u> .	
<u>1.</u>		
<u>1.2</u>		
<u>_1.</u>		
<u>1.4</u>		
<u>1.5</u>		
<u>1.6</u>		
<u>1.7</u>	<u>The structure of this project</u>	4
Chapter 2	- Opening a window and drawing simple graphics with OpenGL	6
<u>2.1</u>	Opening a window using OpenGL	6
<u>2.2</u>	Creating and showing a cube	9
<u>2.3</u>	Difference between flat and smooth shading	11
<u>2.4</u>	Modelling and projection transformations.	15
	- Creating a hierarchical, 3D, wire frame model	
	Building a basic hierarchical model	
<u>3.2</u>	<u>Improving the basic model</u>	
Chapter 4	– <u>Lighting</u>	42
-	Getting started with lighting	
4.2	Colour Tracking	45
4.3	Setting up an object's material properties and shininess	47
	The Material – Lights program.	
4.5	Adding lights to the basic model	57
Chapter 5	- Improving the model: "A more elaborate geometrical example	
	Creating the complex model.	
<u>0.2</u>		
Chapter 6	- Texture Mapping	
<u>6.2</u>		
-	Creating a texture.	
<u>6.4</u>		
<u>Chapter 7</u>	<u>– Conclusions – Future possibilities</u>	83
Appendix	<u>I – Using Borland C++ 5.02</u>	84
Annendiv	II – Using The FLTK Library	87
Appendix	<u>III – Using Paint Shop Pro 5.0</u>	89
Appendix	<u>IV – Bibliography</u>	92

Disclaimer

This document named "A 3D Case Study using OpenGL", was initially written as part of a third year project in the University of Hull, by Fotis Chatzinikos.

This document is free for personal use. For commercial or academic use please contact the Developers Gallery Webmaster at: <u>WebMaster@dev-gallery.com</u>

For updates and the code that accompanies this tutorial please visit the <u>Developers</u> <u>Gallery (www.dev-gallery.com</u>)

Feel free to join our newsletter, so that you will be kept up to date with any additions to the site.

Document Version 1.0

Fotis Chatzinikos, August 29th, 2000

Chapter 1 – Introduction

1.1 About this report

This report was written as part of the final year project with the title "A 3D Case study using OpenGL". In the following pages of this paper, a great deal of information can be found about several different aspects of this project.

Firstly, some information about the style conventions used during the development of this project report is provided. Some background work done mainly in the summer follows. The aims and objectives of this project are the following topic.

Further on, is a short report on what is OpenGL and why it was chosen for the development of this project. The discussions with the supervisor and the time plan of the two semesters also appears here.

Following, a discussion is done on the structure of this project and finally some comments are done on the structure of the accompanying compact disc, which contains all the work done, including this report.

1.2 Style Conventions

In this project report the following style conventions are used :

 \cdot The actual text of the report is written in Arial, size 12. Chapter headings are use Arial, size 20 and secondary heading are of size 16.

- Code is written in Courier New, size 10.
- OpenGL and glut command summaries are shaded with light blue boxes.
- · Variables, arguments, parameter names, etceteras are in *Italics*.
- · OpenGL functions start with 'gl', GLUT functions start with 'glut'.
- Constants of type GL_* or GLUT_* are predefined (OpenGL and GLUT specific).

Note: In the online version of the tutorial not all of the previous apply.

1.3 Background Material

During the summer (before the 5th Semester) some background work was done. This work included searching several Internet sites for information about OpenGL. There are quite a few sites with information on OpenGL but the most useful one proved to be <u>www/opengl.org</u>. At the particular site information is held about OpenGL documentation, specification definitions, developers, example programs, etceteras.

Some information was also needed on human walking in order to make the human model to walk. From Tony White's book on animation [1], the walking cycle of the human model was retrieved.

Some experimentation with OpenGL was done also before the beginning of the 5th semester in order to be familial with the particular graphics system.

1.4 Aims And Objectives

The title of this project is "A 3D Case Study Using OpenGL", so one of the most defined aims of this project is to learn to use OpenGL. What is OpenGL and why it was chosen for the development of this problem are discussed later on. This may be the easiest identifiable aim of the project but is not the most important one.

More important aims are to understand the concept of 3D graphics. People may leave in a three dimensional world but building three dimensional applications is not the easiest thing somebody can do.

Another aim of this project is to learn how to model three dimensional hierarchical objects such as cars, articulated robot arms, humans etceteras. In few words objects with multiple moving parts that are related in some order.

In software engineering terms, a combination of the incremental and prototyping models was used in order to design and work with this project (divide and conquer). This model is based on the idea of constructing a simple and small system as soon as possible, as such a system is probably not complicated; and a simple (not complicated) system is probably correct. As the development of the project is continuing more parts are added to the initial system (Incremental Model). At any points that there is an uncertainty about which algorithm or technique should be used, different solutions can be tried out (Prototyping).

This software engineering model suits the particular project as one of the main objectives of this project is to produce an OpenGL tutorial, something that is clearly incremental.

This technique suits also the developer of this project as he prefers to have something working during the whole development time. Following this technique there was always the drawback of spending more time than designing the whole system and then implementing it, but there was no possibility of reaching the deadline without a working system.

Several Appendices are included with this report. The reason behind these appendices is to keep the length of the main report relatively short, without any loss of information. A short description of the appendices now follows.

 \cdot Appendix I : Using Borland C/C++ 5.02 to build an OpenGL DOS console WINDOWS program.

• Appendix II: Using the FLTK (Fast Light Tool Kit) GUI (Graphical User Interface) to construct buttons, dialogs, menu etceteras in C.

- Appendix III: Using Paint Shop Pro to retrieve the model data
- · Appendix IV: Bibliography

1.5 What is OpenGL?

According to the OpenGL data sheet, OpenGL is an industry standard, stable, reliable and portable, evolving, scalable, easy to use and well-documented API.

But lets explain all these buzzwords. OpenGL is an industry standard (by now) as it has been available from 1992. The OpenGL specification is managed by an independent consortium, the OpenGL Architecture Review Board, some of its members being SGI (Silicon Graphics) and Microsoft.

OpenGL is available for more than seven years in a variety of systems. Additions to the specification (through extensions) are well controlled by the consortium and proposed updates are announced in time for developers to adopt changes. Backwards compatibility is also ensured.

Chapter 1 – Introduction

OpenGL is reliable as all applications based on OpenGL produce consistent visual display results on any OpenGL API compliant hardware. Portability is also a fact as OpenGL is available in a variety of systems, such as PCs, Macintoshes, Silicon Graphics and UNIX based machines and so on. OpenGL is available also in different bindings, some of them being C and C++, Java and FORTAN.

OpenGL is evolving through its extensions mechanism that allows new hardware innovations to be accessible to the API, as soon as the developers have the hardware (and the extension) ready.

OpenGL is also scalable as it can run in a variety of computers, from 'simple' home systems to workstations and supercomputers. This is achieved through OpenGL's hardware capabilities inquiry mechanism.

OpenGL is well structured with logical commands (a few hundred). OpenGL also encapsulates information about the underlying hardware, freeing the application developer from having to design hardware specific code.

OpenGL's data sheet says that numerous books are available on the subject. Actually at the start of 1998 only two of them were widely available, the *OpenGL programming guide* [3], and *the OpenGL Super Bible* [4]. The truth is that at the end of 1998 a few more books appeared about OpenGL and that the World Wide Web has enough resources available for free.

Lets see now the programmers view of Open. To the programmer OpenGL is a set of commands. Firstly he opens a window in the frame buffer into which the program will draw. After this some calls are made to establish a GL context. When this is done, the programmer is free to use the OpenGL commands to describe 2D and 3D objects and alter their appearance but changing their attributes (or state). The programmer is also free to manipulate directly the frame buffer with calls like read and write pixels.

So why was OpenGL chosen for the development of this project. Why OpenGL instead of DirectX, GKS or XGKS ?

The answer is a combination of the just mentioned OpenGL advantages and the fact that DirectX is quicker (only at the moment) but OpenGL is more precise and that GKS (and X–GKS) are years in the market but are not as platform independent as OpenGL (OpenGL is also free). And for scientific visualisation, virtual environments, CAD/CAM/CAE, medical imaging and so on (not just games) precision and platform independence are the key features.

1.6 Discussions With The Supervisor-Time Plan

Supervisor meeting took place every Monday during the first semester. At each meeting the supervisor was told of new developments during the previous week. These developments were discussed and on one occasion a change in the program was made (the program made to read data from a file). Another decision that may be of some importance is, that it was agreed that after finish the modelling of the human (chapter 4) no further improvements were to be done, until texture mapping (the next chapter) was finished.

1.7 The structure of this project

The structure of this project is such that a newcomer to three–dimensional graphics and OpenGL can follow easily, building up knowledge before moving on to more complicated concepts, in other words this project is written in the form of an OpenGL tutorial.

The topic of the second chapter is simple window construction, as OpenGL needs a graphical (windowing) operating system, and the introduction of modelling and projection transformations. The reader first learns how to open windows using OpenGL, and then a discussion follows on modelling transformations like rotation, scaling and translation and projection transformations like orthographic and perspective viewing.

Chapter 1 – Introduction

In the third chapter a first attempt is made to create a simple model of a man and the appropriate animation cycle, which resulted in a model constructed from basic geometrical shapes (spheres and cubes). Previously acquired knowledge of modelling and projection transformations is used in order to construct this simple model. At first the example programs use data that are 'hard–wired' in the program, but latter on the examples become data–driven.

The model of a man was chosen because it is an interesting case. Firstly, it is a hierarchical model, meaning that there are many interrelations between certain parts of the body such that cause the rotation and movement of body parts when other, higher in the hierarchy parts are moving. Secondly, the animation of a human walk cycle is a very interesting topic that is still a research topic. In this project a simple, but effective animation technique was chosen, based on the idea of 'key–framing'.

The fourth chapter introduces OpenGL's lighting model and continues with a discussion on materials and their properties. A material is an object property approximating real–life materials. When proper material components are chosen, material like wood, glass, steel, etc. can be constructed. A program is constructed where a user can experiment with the light and material properties in order to familiarise with the concept.

A more elaborate geometrical example is presented in the fifth chapter, as its discussion topic is the improvement of the basic, model of a man. A three–dimensional model of a man is created, using a technique that is able to construct a three–dimensional model from several two–dimensional images.

The sixth chapter introduces texture mapping, a technique that enables the use of images as parts of objects, making OpenGL programs more attractive and 'real'. The topic of texture mapping is not going to be throughoutly exhausted, as the subject is quite complicated and the applications of texture mapping are inexhaustible. An example program is created that can load a bitmap image, select a part of it, in order to create a texture and then this texture is applied on a rotating cube and then on the improved model of a man itself. The user can interactively set different texture properties like texture filters and so on. A function that is able of saving a texture as a bitmap file is also available.

The seventh, and final chapter contains the conclusions of this report and future possibilities that arise from this project.

As mentioned in the first chapter, OpenGL programs need a graphical window interface in order to work, possibilities include Microsoft's Windows systems, Silicon Graphics systems and X–Windows systems. In this chapter introductory material of OpenGL will be discussed, things like opening and naming a window, clearing the window and drawing simple graphics like a cube.

The first example demonstrates how to open a window by using the *GL Utility Toolkit* named *GLUT*. This library will be used quite often as it contains many functions that without them simple *OpenGL* programs would be quite tedious to write.

The second program goes a bit further and demonstrates how to create and show a cube using *OpenGL*. At this point the cube looks two-dimensional as the projection used is orthographic (projections are described in detail later).

The third example expands the second one, in order to show the difference between flat and smooth shading. A square is drawn either by using flat or smooth shading. The user is able to change back and forth interactively in order to see the difference.

The fourth program draws four cubes. Two of them are displayed with orthographic projection and two with perspective projection. Different order of translation and rotation is also applied in order to demonstrate the different effect.

2.1 Opening a window using OpenGL

The goal of this section is to create an *OpenGL*-based window. There are many ways in which a window can be created and shown under the various windowing systems, but *OpenGL's Utility Toolkit, GLUT*, provides some functions that can create a window in an Operating System independent way. This means that programs created with *GLUT* will operate under different windowing systems without having to change the code manually.

In order to use *OpenGL* and *GLUT* the header file glut.h is needed. This file contains references also to the header files opengl.h and glu.h. These three files are all that is needed at the moment in order to construct some simple *OpenGL* programs. The file windows.h is also need to be included <u>before</u> the inclusion of the *OpenGL* header files, otherwise the compiler will give quite a few errors. In order to make the program portable, the following piece of code can be written (as the file windows.h will not be needed for example with a Silicon Graphics machine).

Example 2.1 Checking for execution platform type

```
#ifdef __FLAT__
#include windows.h
#endif
```

This will check at compile time if the environment is a Microsoft's Win32 environment (Windows 95/98/NT) and if the check is true the file will be included, otherwise the file will not be included (in X–Windows for example).

A window has several properties, like dimensions, name, buffers and so on. These properties must be initialised before the actual window is created and shown. *GLUT* provides several functions for this particular reason. In this example the calls to these functions can be found inside the body of the **main** function. The initialisation of the window is the topic of the next paragraph.

Before using any *GLUT* functions the *OpenGL Utility Toolkit*, *GLUT* must be initialised. This is done by calling the function **glutInit** inside the main function of the program. After *GLUT* is initialised the display mode of the window must be initialised, too. Calling the function **glutInitDisplayMode** will do the last. This function accepts quite a few arguments as the display mode of a window can be double or single buffered, RGB or indexed colour table, with or without a depth buffer etc.

The next thing to do is to call the function **glutCreateWindow** in order to create the actual window, but prior to that, the two functions **glutInitWindowSize** and **glutInitWindowPosition** must be called. The first one as its name implies is responsible for setting the size of the window and the second one for setting the window's initial position. Both size and position can change later on. The last two functions accept two integer arguments, each specifying pixel dimensions. In the case of **glutInitWindowSize** the arguments are its width and height. In the second case the two arguments are the horizontal and vertical distance from the upper left corner of the monitor, where the window in creation should appear (if possible). The function **glutCreateWindow** accepts a string as its argument. This string will be used as the window's name.

Now that the window is actually created, only a few steps remain before the window is ready and visible.

In the following lines, the code that is responsible for doing all the previous operations can be seen (Example 2.2).

Example 2.2 Code to initialise and create a window

```
int main(int argc, char** argv)
{
    glutInit (&argc, argv) ;
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB) ;
    glutInitWindowSize (400, 100) ;
    glutInitWindowPosition (100, 100) ;
    glutCreateWindow ("First Chapter - Opening an OpenGL Window") ;
    init() ;
    glutDisplayFunc (display) ;
    glutMainLoop () ;
    return 0 ;
}
```

In this piece of code the actual calls to the *GL* functions can be seen, in order to create a RGB, double buffered window that has 400 pixels width, 100 pixels height and is named "*First Chapter – Opening an OpenGL Window*". This window will be positioned (if possible) 100 pixels from the upper left corner of the screen (both horizontally and vertically). Some other functions are visible here that have not been mentioned before.

The function **init** is responsible for any initialisation needed prior to the window construction and/or visualisation. Its structure can be seen here.

Example 2.3 The init function

```
void init(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0) ;
    glShadeModel(GL_FLAT) ;
}
```

This function contains just two OpenGL calls. The first one, named **glClearColor** is responsible for setting the initial clearing colour. The clearing (background) colour in this occasion is set to white (all colour values,

Red, Green and Blue are set to one). The fourth value (0.0) is the one called alpha value and is normally used for blending. At this point the alpha value is of no importance.

Next the function **glShadeModel** is called in order to set the shading model. The shading model can be either GL_SMOOTH or GL_FLAT. When the shading model is GL_FLAT only one colour per polygon is used, whereas when the shading model is set to GL_SMOOTH the colour of a polygon is interpolated among the colours of its vertices. An example will demonstrate this particular difference later on.

Back in the **main** function, two more calls follow the call to the function **init**. The first one, named **glutDisplayFunc**, is the first and most important event callback function that will appear in this report. The callback functions are special functions that are registered in order to do some specific operations. Whenever *GLUT* determines that the contents of a window need to be redisplayed, the callback function registered by **glutDisplayFunc** is executed. Therefore all the code that has to do with drawing must be inside the display callback function.

The following code shows the function **display**. **display** is the function registered as the display callback by calling the function **glutDisplayFunc**(**display**).

Example 2.4 Basic display function

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT) ;
   glutSwapBuffers() ;
}
```

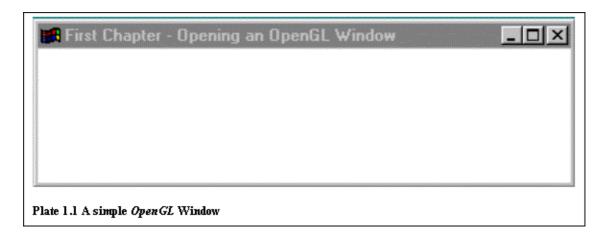
Like all the functions that will be used as display callback functions, **display** is of type void. As this program is quite simple and no actual drawing happens in the window, the contents of this function are quite simple, too. They are actually the only "compulsory" function calls that should always appear in any display callback function.

The first one, **glClear** must be called prior to any drawing as it clears the background. It can be omitted if it is desired to draw several times without clearing the background! It accepts one argument that specifies the desired buffer to be cleared. In this program, as no actual drawing happens, this function just clears the background to the colour set previously in the **init** function by the function **glClearColor**.

The function **glutSwapBuffers** does exactly what its name implies. It swaps the back buffer with the front buffer, as when a window is double buffered, the default drawing buffer is the back buffer. Any actual drawing happens in the back buffer and when the drawing is ready the two buffers are swapped in order to achieve smoothness and remove any flickering. If a window had only a single buffer the call **glFlash** would be used instead.

Back in the **main** function the last routine called is **glutMainLoop**. After all setup is done, *GLUT* programs enter this event–processing loop, never to exit until the program is finished.

The results of the program, after compiling, linking and running can be viewed in plate 2.1. Further information on how to make an *OpenGL* project in Borland C/C++, compile and link, can be found in Appendix I.



2.2 Creating and showing a cube

Now that it has been demonstrated what must be done to create and show a simple *OpenGL* window, it is the time to go a bit further and create a simple cube.

In this section the previously acquired knowledge is going to be used in order to create a window. When the window is created, it is then quite easy to draw a simple wireframe cube just by calling some OpenGL functions. The steps needed to create such a simple cube will be the topic of this section.

As the following programs (in the next chapters) will be quite complicated, a first attempt will be made in this program to try and create a project that its code will be separated in more than one files (in order to keep the code simple and easy to understand). For the purposes of this example only three files will be needed. The first one will contain the main program and is named *main.c*, the second one is called *model.c* and will contain the functions that will be responsible for drawing any models, in this case a simple function that draws a wireframe cube of constant size. The last file is named *model.h* and it is the header file that will contain any function definitions needed by the main program. These functions will be implemented in the *model.c* file.

This program is mainly the same as the previous one with the only additions being a slight change of the **display** function in order to draw a wireframe cube and the splitting of the project in three different files.

Example 2.5 Display function that draws a wireframe cube

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT) ;
    Draw_Wireframe_Cube() ;
    glutSwapBuffers() ;
}
```

As it can be seen in Example 2.5 the only difference from the **display** function in the previous program (Example 2.4) is the inclusion of the function **Draw_Wireframe_Cube**. This function is responsible for creating a simple wireframe cube of size one. This function is available to the main program by including the header file *model.h*. The structure of this file can be seen in example 2.6.

Example 2.6 Basic header file

```
#ifndef MODEL
#define MODEL
#ifdef __FLAT___
#include <windows.h>
#endif
```

```
#include <gl/glut.h>
void Draw_Wireframe_Cube(void) ;
#endif
```

This header file contains the definition of the function **Draw_Wireframe_Cube**. It can be seen that this function is of type void and that it does not accept any parameters. The implementation of this function can be found in the file *model.c* and example 2.7 shows the contents of this file. The statement *#ifndef* is used for conditional compilation and compilation time minimisation. When the compiler tries to compile the particular file, it checks if the file is already defined. If the file is not defined, then it continues compiling, otherwise it does not compile the file. For example if a particular header file is referenced from several implementation files, and the compiler has already compiled the particular header file (and named it using the *#define 'identifier'* statement) there would be no reason to recompile it.

Example 2.7 Basic implementation file

```
#include "model.h"
void Draw_Wireframe_Cube (void)
{
    glColor3f(0.0,0.0,0.0);
    glutWireCube(1.0);
}
```

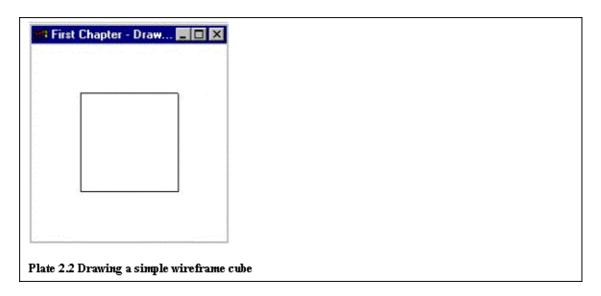
It can be seen in this example that an implementation file must include its definition file (in this case *model.h*) and of course the implementation of the functions defined in the header file. When including files, <> are used to direct the compiler to look for the particular file in the systems directory and "" are used to direct the compiler to look in the current directory for the specified file.

In example 2.7, two new *GL* functions are introduced (an *OpenGL* and a *GLUT* function). The *OpenGL* function named **glColor3f** is responsible for setting the current colour. As the working colour mode is RGB (Red–Green–Blue), this function accepts three parameters; one for the red, one for the green and one for the blue value of the colour. The values of these parameters can range from 0.0 to 1.0 (black being zeros for all red, green and blue parameters and white being ones for all three parameters). In this example the colour is set to black (0.0, 0.0, 0.0).

The *GLUT* function named **glutWireCube** is responsible for drawing a wireframe cube that its size is specified by its one, floating point, parameter. In this example the size of the wireframe cube is set to one. So the function **Draw_Wireframe_Cube** just sets the colour to black and draws a wireframe cube.

At this point the program is ready. If it is compiled and run the results will be the ones shown in plate 2.2.

The cube looks like a simple rectangle because the default *OpenGL* projection is orthographic (more on projections in section 2.4), so only the front face of the cube is visible and in such a way that it hides the five remaining faces.



2.3 Difference between flat and smooth shading

In this example the difference between flat and smooth shading will be demonstrated. In this case a square will be drawn, but without using the GLUT function **glutWireCube** (or **glutSolidCube**). These *GLUT* functions will not be used, as they do not provide any way of setting different colours to different vertices, and in order to demonstrate smooth shading at least two of the vertices of a polygon should be of a different colour. It would be easier to create a rectangle by using the function **glutSolidCube** and then scaling it down in order to make it flat (a flat cube is a square), but as it was just mentioned this can not be done (because of the colours). A custom function will be created that will draw a square with four different colours assigned to each of the four vertices. Example 2.8 demonstrates this function called **Draw_A_Rectangle**.

Example 2.8 Function Draw_A_Rectangle

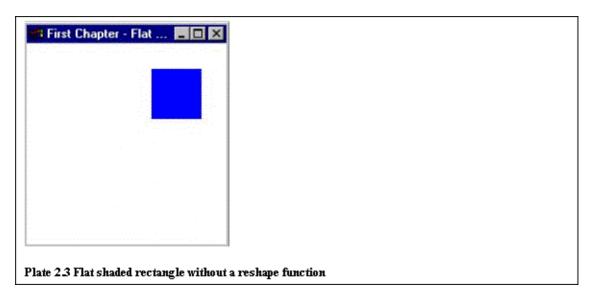
```
void Draw_A_Rectangle(void)
{
    glBegin(GL_QUADS) ;
    glColor3f(0.0,1.0,0.0) ;
    glVertex2f(0.25,0.25) ;
    glColor3f(1.0,1.0,0.0) ;
    glVertex2f(0.25,0.75) ;
    glColor3f(1.0,0.0,0.0) ;
    glVertex2f(0.75,0.75) ;
    glColor3f(0.0,0.0,1.0) ;
    glVertex2f(0.75,0.25) ;
    glEnd() ;
}
```

A square has four vertices and as seen in example 2.8 only four calls to the function glVertex2f are needed. The function glVertex2f specifies two-dimensional vertices. Other graphics systems need an extra vertex in order to 'close' a shape; *OpenGL* does not need this extra call as when glBegin is called with the parameter GL_QUADS, *OpenGL* automatically connects the first and the fourth vertices. When a shape (a geometric primitive) is constructed in *OpenGL*, it is always bracketed between the commands glBegin and glEnd.

Between **glBegin** and **glEnd** several different OpenGL commands can be issued. In this example only two different ones are used; **glColor3f** to set the current colour and immediately afterwards **glVertex2f** to specify a vertex of the previously set colour. By passing the particular values in the four **glVertex2f** commands, a rectangle that lies from 0.25, 0.25 to 0.75, 0.75 is created. With its four vertices having the colours green, yellow, red and blue (from left to right).

If the previous program (demonstrated in section 2.2) is slightly changed and instead of using the function

Draw_Only_Cube uses the function **Draw_A_Rectangle** (inside the display function), the result will be the one shown in Plate 2.3. The square will appear blue and at the upper right corner of the window.



The square appears blue because the shading mode was set to GL_FLAT (in example 2.3); that means that only one colour per polygon is used. Later on this example the effect of smooth shading will be shown. At the moment, the concentration will be on why the rectangle appears on the upper right corner of the window. This happens because the default projection mapping of *OpenGL* is orthographic and has boundaries from -1 to 1 in all three dimensions. An orthographic projection can be thought as a 3D rectangle. This results in the showing of the rectangle in the upper right corner of the window, as the centre of the window has the co–ordinates 0,0 and the upper right corner the co–ordinates 1,1 (on the X, Y axes). If the rectangle needs to be shown in the centre of the screen, a means of manipulating the projection area has to be found. Introducing the function **glutReshapeFunc** can do this.

This is another *GLUT* callback function, quite similar to the one described before named **glutDisplayFunc**. This function specifies the function that will be called whenever the window is resized or moved. It can also be used to initialise the projection type. Example 2.9 shows the reshape function for the particular program.

Example 2.9 Reshape function that specifies a 2D area (0,0 to 1,1)

```
void reshape(int w, int h)
{
    glViewport ( 0, 0, (GLsizei)w, (GLsizei)h) ;
    glMatrixMode(GL_PROJECTION) ;
    glLoadIdentity() ;
    gluOrtho2D(0.0,1.0,0.0,1.0) ;
    glMatrixMode(GL_MODELVIEW) ;
    glLoadIdentity() ;
}
```

In this function a few new *OpenGL* functions are used. The first one, **glViewport** is responsible for setting the current window's viewport. A viewport specifies the part of the window that all the drawing will take place, with parts out of the viewport normally clipped out. In this case the viewport will be the whole window as the values that are passed to the function **glViewport** specify the viewport to lie from point (0, 0) and for w pixels width and h pixels height. w and h are the width and height of the current window, so the viewport is the whole window.

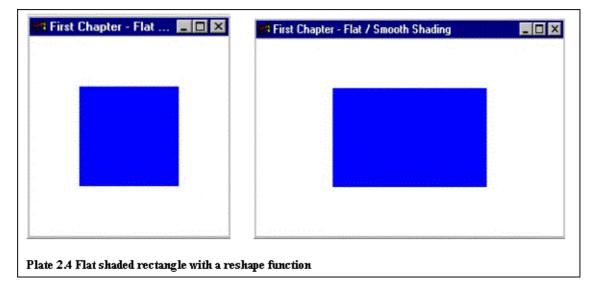
The next call is to the function **glMatrixMode**. This function is responsible for setting the current matrix mode, as in *OpenGL* more that one mode exists. As it is seen the argument to the first call of **glMatrixMode** is GL_PROJECTION; this means that any matrix manipulations from this point onwards will

affect the projection matrix. A couple of lines later the same routine is called, but this time the argument is GL_MODELVIEW. This indicates that succeeding transformations now affect the modelview matrix instead of the projection matrix.

In example 2.9 after a call to the routine **glMatrixMode** (in both occasions), a call to the routine **glLoadIdentity** follows. This routine is responsible for clearing the current modifiable matrix from any previous transformations by setting it to the initial identity matrix.

The function that is responsible for setting the projection area of the window follows. As previously noticed, the default *OpenGL* projection is 3D orthographic with boundaries from -1 to 1 in all three dimensions. The routine **gluOrtho2D** is used to transform the projection to two-dimensional (by setting the z boundaries to -1 and 1). The clipping boundaries are specified with four arguments that the routine accepts. In this case the 2D orthographic projection area is set to be from (0, 0), the lower left corner of the window to (1, 1) being the upper right corner of the window.

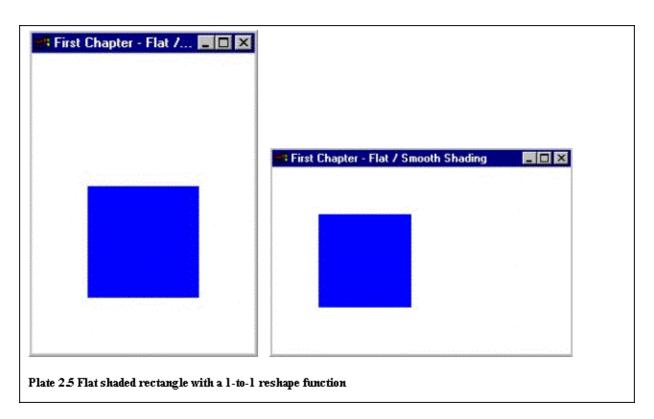
If the **main** function is slightly modified in order to include a call to the routine **glutReshapeFunc** with the argument being **reshape** (just after the call to **glutDisplayFunc**), the results from the program will be the ones shown in Plate 2.4.a. The problem is that if the window is slightly reshaped (Plate 2.4.b) the rectangle will also be



reshaped (it will stop being a square).

This can be changed so that the rectangle will always appear as it was initially set. Example 2.10 demonstrates the slight change to the **reshape** function in order to accommodate that.

Example 2.10 Reshape function that



This code finds which of the two sides (height or width) is longer and then sets the viewport in such a way that it is always square (either h x h or w x w). When the new program is compiled and run the results are the ones shown in Plate 2.5.

Now that everything about the projection used is explained, it is time to go on and see what happens if the argument passed to **glShadeModel** changes from GL_FLAT to GL_SMOOTH. This time the rectangle will not appear blue but its colour will be calculated by interpolating the colours of its four vertices. Plate 2.6 shows exactly that.

This example also gives an opportunity to introduce keyboard interaction, by introducing the GLUT routine **glutKeyboardFunc**. This function is similar to **glutDisplayFunc** and **glutReshapeFunc**, as it is used to register a keyboard callback routine. Example 2.11 shows the structure of the keyboard function that can change between flat and smooth shading by using the keys 'f'-'F' (for flat shading) and 's'-'S' (for smooth shading).

Example 2.11 The keyboard function

```
void keyboard (unsigned char key, int x, int y)
{
   switch (key)
   {
      case 's' :
      case 'S' :
        glShadeModel(GL_SMOOTH) ;
        break ;
      case 'f' :
      case 'F' :
         glShadeModel(GL_FLAT) ;
         break ;
      default :
         break ;
   }
  glutPostRedisplay() ;
```

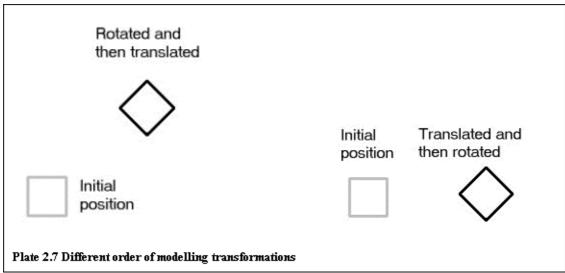
}

A new *GLUT* routine is also introduced in this function, **glutPostRedisplay**. This routine marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by **glutDisplayFunc** will be called to redraw the window. If the routine **glutPostRedisplay** was not included at this point, the user could press keys without receiving any feedback at all. This happens because *OpenGL* does not know that the contents of the window change whenever the user presses a key, as it is quite normal that the keyboard will be used for reasons other than changing the contents of the window.

lat 🔳 🛛 🗶	
led rectangle	

2.4 Modelling and projection transformations

This section will try to explain the main modelling transformations, translate, rotate and scale and the basic projection transformations, orthographic and perspective projection. In order to achieve that, four cubes will be drawn in the four quadrants of the window. The upper two cubes will be shown using orthographic projection; whereas the lower two by using perspective projection. Different order of modelling transformations will be used to demonstrate this particular difference.



Modelling transformations are used to position and orient the models. Three basic transformations are available in *OpenGL* and these are translation, rotation and scaling. The order of these transformations is not irrelevant to the final transformation. For example, if an object is firstly translated and then rotated, it will have a different position and orientation from the same object that has been firstly rotated and then translated. This particular difference will be demonstrated later in the example program, but for now this difference can be viewed in Plate 2.7. In the picture on the left, a cube is firstly rotated 45 degrees and then translated x

units. In the picture on the right, the same cube is firstly translated the same x units and then rotated by 45 degrees. After having introduced the concept of modelling transformations, the concept of projection transformations will follow.

Specifying the projection transformation is like choosing a lens for a camera. This transformation can be thought as choosing the field of view (FOV) or viewing volume and therefore what objects are inside and how they look. Back to the camera example, it is like choosing among different lenses. With a wide–angle lens, a bigger area is included in the photo than with a normal or telephoto lens, but with a telephoto lens more detail appears in the photograph, as objects look nearer. In computer graphics zooming in and out of an object is much easier than changing lenses in a camera, as the only thing to be done is to choose a smaller field of view.

In addition to the field of view considerations, the projection transformation determines how objects project (look) on the screen. Two types of projections are provided with *OpenGL*, perspective and orthographic projection.

The first type of projection, perspective projection, matches how objects appear in real life. Perspective makes objects that are further away appear smaller; for example it makes the two sides of a road appear to converge in the distance. For realistic looking pictures, this type of projection should be used.

Orthographic projection on the other hand, maps objects directly on the screen, without altering their relative size. This type of projection is useful for many CAD–based applications like circuit design or architectural planning, as the user needs to see actual measurements of objects, rather than how these objects look. Architects can use perspective projection in order to visualise how a particular building or room would look from a particular viewpoint, and then switch to orthographic projection in order to print out the blueprint plans.

Now that the theory of modelling and projection transformations is partially explained, the actual example that demonstrates these transformations can follow.

This project is split into four different implementation files and their corresponding header files (except the main program that does not have a special header file). The file main.c contains the main program, the file model.c contains the modelling routines, the file transformations.c contains the modelling transformation routines and the file keyboard.c contains the keyboard interaction routines.

Starting by the main program it can be noted that there is no need for a **reshape** function as all the operations that would normally occur in the body of such a function are carried out in the body of the **display** function. A two-dimensional array of type float is used in order to communicate the rotation, translation and scaling values in the four different files. Actually this array is needed in the transformations file, so it was declared in the header of this file as **#extern**.

In the body of the **init** function (example 2.11) this array is partially initialised. The remaining array elements are not initialised here, as there is no need to do so. The constants SCALE and ROTATE are declared in the file model.h as 2 and 1 (TRANS is also declared there as 0). The scaling elements are initialised to 1 (no scaling) and the rotate elements of the array are initialised to zero, except the first one that is initialised to 1 (initially rotate only the x-axis).

Example 2.12 Init function

```
void init(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0) ;
    glShadeModel(GL_FLAT) ;
    tran[SCALE][0] = 1.0 ;
    tran[SCALE][1] = 1.0 ;
```

Chapter 2 - Opening a window and drawing simple graphics with OpenGL

```
tran[SCALE][2] = 1.0 ;
tran[ROTATE][0] = 1.0 ;
tran[ROTATE][1] = 0.0 ;
tran[ROTATE][2] = 0.0 ;
```

}

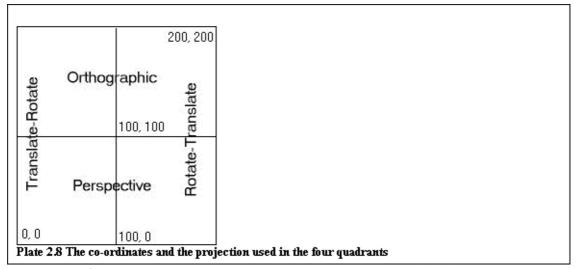
The **main** function of the program is the same as the one in the previous program with the only difference being that there is no **glutReshapeFunc**, as a **reshape** function is not needed. A function that is quite different (from the previous program) is the **display** function, as this is the place where the actual drawing and placing of the four cubes that constitute this example happens. Example 2.12 contains the code of this function.

Example 2.13 display function that positions and draws four cubes

```
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT) ;
  glShadeModel(GL_FLAT) ;
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  glViewport(0,125,125,125);
  glOrtho(-2.0,2.0,-2.0,2.0,2.0,-2.0);
  glMatrixMode(GL_MODELVIEW) ;
  Draw_Cube_Transl_Rot() ;
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  glViewport(0,0,100,100) ;
  gluPerspective(60.0, 1.0,1.0,20.0) ;
  glTranslatef(0.0,0.0,-4.0) ;
  glMatrixMode(GL_MODELVIEW) ;
  Draw_Cube_Transl_Rot() ;
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  glViewport(100,100,100,100) ;
  glOrtho(-2.0,2.0,-2.0,2.0,2.0,-2.0) ;
  glMatrixMode(GL MODELVIEW) ;
  Draw_Cube_Rot_Transl()
  glMatrixMode(GL_PROJECTION) ;
  glLoadIdentity() ;
  glViewport(100,0,100,100) ;
  gluPerspective(60.0, 1.0,1.0,20.0) ;
  glTranslatef(0.0,0.0,-4.0) ;
  glMatrixMode(GL_MODELVIEW)
  Draw_Cube_Rot_Transl() ;
  glutSwapBuffers() ;
}
```

This function contains four very similar parts. All of them start by setting the projection matrix as the current. A call to **glLoadIdentity** follows in order to initialise the matrix and then a call to **glViewport** is done in order to set the viewport. The arguments to the **glViewport** routine are such that will divide the window (of 200 by 200 pixels) in four equal quadrants. After this is done a call to either **glOrtho** or **gluPerspective** is done in order to set the projection of the current quadrant. Plate 2.8 shows the co–ordinates of the four quadrants and their projection.

Chapter 2 - Opening a window and drawing simple graphics with OpenGL



The call to **glOrtho** and its arguments are quite easy to understand, as it is similar to the routine **gluOrtho2D** that has already been explained (the only difference is that **glOrtho** sets also the z boundaries). The routine **gluPerspective** needs further explanations. This routine creates a symmetric perspective–view frustum (a three–dimensional area). The first argument to the routine is the angle of the field of view (in the x–z plane). The second argument is the aspect ratio of the frustum (normally its width divided by its height) and the remaining two arguments are the near and far values of the frustum. These values the distances between the viewpoint and the clipping planes along the negative z–axis, and they should always be positive. As the frustum will always lie on the negative z–axis the object shown most times will be needed to be translated some value x before shown to the screen. This is why a call to **glTranslatef**(0.0, 0.0, –4.0) is necessary after a call to **gluPerspective**. The correct amount of translation and frustum creation is not something that can be shown exactly but comes naturally after becoming 'comfortable' with the concept.

When this is done a call to **glMatrixMode**(GL_MODELVIEW) resets the current matrix to the modelview matrix in order to draw the four cubes. After the matrix is set to modelview a call to either **Draw_Cube_Transl_Rot** (for the two quadrants on the left–hand side) or **Draw_Cube_Rot_Transl** (for the two quadrants on the right–hand side) is done. This is done because these two functions that both draw a cube, contain calls to the routines **glTranslate** and **glRotate**. As explained before the order of these routines is important. To demonstrate this, the routine **Draw_Cube_Transl_Rot** draws the cube after applying the transformations in the order of translate and then rotate, whereas the second one draws the cube after applying these two transformations in the opposite order, in order to visualise the difference.

At this point the main program is ready. The two custom functions that were used, **Draw_Cube_Transl_Rot** and **Draw_Cube_Rot_Trans** are available in the program by including the header file keyboard.h as this file contains references also to the files model.h and transformations.h.

This section will be continued by examining these two custom functions whose implementation is in the file transformations.c. Example 2.13 contains the code of the first one, **Draw_Cube_Transl_Rot**. The code of the second one is the same with the only difference being that the order of the routines **glTranslate** and **glRotate** is the opposite.

Example 2.14 display function that positions and draws four cubes

```
void Draw_Cube_Transl_Rot (void)
{
    glPushMatrix() ;
    glTranslatef (tran[TRANS][0],tran[TRANS][1],tran[TRANS][2]) ;
    glRotatef (tran[ROTATE][3], tran[ROTATE][0], tran[ROTATE][1],tran[ROTATE][2] ) ;
    glScalef(tran[SCALE][0],tran[SCALE][1],tran[SCALE][2])
```

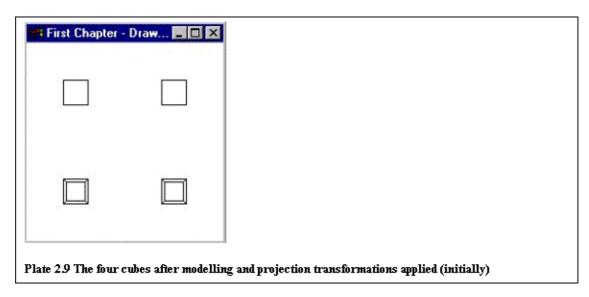
Chapter 2 - Opening a window and drawing simple graphics with OpenGL

```
Draw_Black_Cube() ;
glPopMatrix() ;
}
```

It is noticeable that two statements named **glPushMatrix** and **glPopMatrix** wrap the body of the function. The routine **glPushMatrix** is responsible for saving the current matrix in the matrix stack. The matrix stack is a stack that is used to save and restore transformation matrices during the execution of an *OpenGL* program. Actually there are three different matrix stacks; one for modelling transformations, one for projection transformations and one for texture transformations. This means that a particular matrix can be saved in the stack by using **glPushMatrix**, transformations that modify the matrix can be done and when the previous to the modifications matrix is needed again, it can be loaded or 'popped' from the stack by using the command **glPopMatrix**. These two routines are particularly helpful when building hierarchical models and they are going to be explained in detail when time comes.

Between these two calls four more routines are called, a **glTranslate**, a **glRotate** and a **glScale**.

glTranslate and **glScale** accept three arguments being the X, Y, and Z values the object should be translated or scaled. In the case of **glScale** an argument of 1 means no scaling, an argument of 0.5 means reduce the scale in half and an argument of 2 means double the scale. **glRotate** accepts four arguments with the first one being the amount of degrees the object should be scaled and the other three varying from 0 to 1. If 0 is passed the particular axis is not rotated, whereas if 1 is passed the particular axis is fully rotated. The arguments of these routines (elements of the *tran* array) are modified externally by other functions that will be explained shortly. After all modelling transformations are done, a call to **Draw_Black_Cube** is made in order to draw a transformed (due to the modelling transformations) black cube. The routine **Draw_Black_Cube** can be found in the file model.c and it is a very simple routine as it just sets the current colour to black and uses the routine **glutWireCube**(1.0) to draw a cube of size 1.0.



At this point if the program is compiled and run the results will be the ones shown in Plate 2.9. The difference between perspective and orthographic projection is clearly visible, but the difference in the order of the application of the modelling transformations is not yet visible, as none of them have been applied yet.

The file keyboard.c contains the **keyboard** function that is needed in order to interact with the program through the keyboard. This function is shown in example 2.15.

Example 2.15 display function that positions and draws four cubes

```
void keyboard (unsigned char key, int x, int y)
{
    switch(key)
```

```
{
   case 'a' :
     Rotate_Cube() ;
      glutPostRedisplay() ;
      break ;
   case 's' :
     Rotate_Cube3D() ;
      glutPostRedisplay() ;
      break ;
   case 'd' :
      Move Cube() ;
      glutPostRedisplay() ;
      break ;
   case 'f' :
      Scale_Cube() ;
      glutPostRedisplay() ;
      break ;
   case '1' :
      glutIdleFunc(Small_Anim) ;
      break ;
   default :
      glutIdleFunc(NULL) ;
      break ;
}
```

}

After examining example 2.14 it is clear that by pressing the keys a, s, d, f and 1 five different things will happen. These five functions will be explained shortly. Something new to this function is the call to the function **glutIdleFunc**. This function can be called in order to do something when the program is idle, for example a small animation. The effect of this function becomes inactive when a NULL is passed to it. As it is seen in the example if the user presses any other than the specified keys, a call to **glutIdleFunc** is done with a NULL argument in order to stop any previously issued **glutIdleFunc** routine.

Back in Example 2.10, a single call to **glutPostRedisplay** was issued just after the end of the switch statement. This approach is not followed here because it is not needed to issue a **glutPostRedisplay** command every time a key is pressed. However, in the case of the key '1' the call to **glutPostRedisplay** must be inside the function **Small_Anim**.

These five functions are part of the file transformations.c. These functions will be explained here but their code will not appear as they are quite simple to understand. The first one, **Rotate_Cube3D** sets the first three rotation elements of the *tran* array to 1, and the fourth one is incremented by 1 each time the function is called. This is done because of the structure of the **glRotate** function. As these array elements are used in a call to **glRotate** of the same form used in example 2.13 and the function is required to rotate all three axes, the last three arguments to the **glRotate** function should be 1 (tran[0] to tran[2]) and the first argument should contain the degrees of the required rotation (tran[3]).

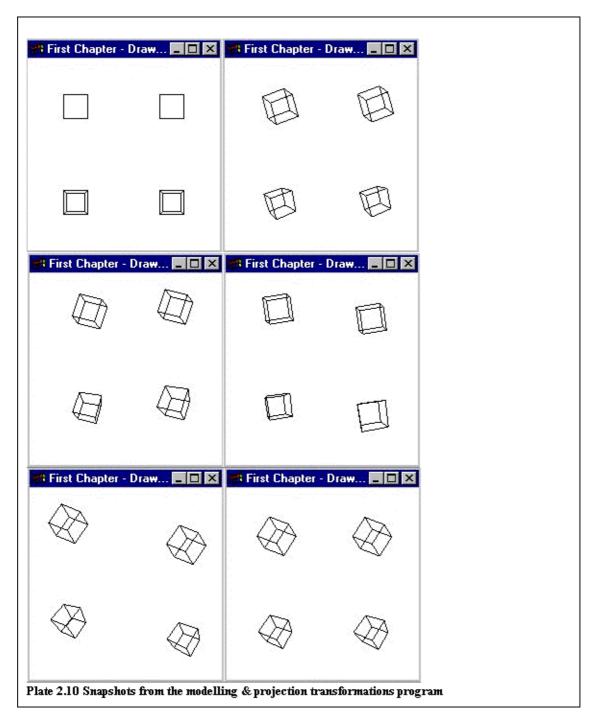
The second function, **Rotate_Cube** gradually rotates first the x-axis for 85 degrees, then the y-axis for the same degrees, then the z-axis for the same amount of degrees and finally continuously rotates all three axes.

The third function, **Move_Cube** does some translation transformations that result in the movement of the cube in a square pattern (rightwardsà upwards à leftwards à downwards à rightwards à and so on).

The fourth function, **Scale_Cube** scales the cube up and down between twice its original size and a quarter of it.

The remaining function, **Small_Anim** uses the previously defined functions **Rotate_Cube3D** and **Move_Cube** in order to demonstrate the difference in the application of the modelling transformations. The results of this function will be different when using the functions **Draw_Cube_Trans_Rot** and **Draw_Cube_Rot_Trans** to visualise the cube as it simultaneously translates and rotates the axes.

If the program is compiled and run, and the user presses the key '1' to invoke the function **Small_Anim**, the results will be the ones shown in Plate 2.10. The difference between both the order of applying the modelling transformations and the projection transformations is now clearly visible.



<u>Chapter 3 – Creating a hierarchical, 3D, wire frame</u> model

Now that the basic *OpenGL* and *GLUT* structure and routines has been explained, it is time to go on and try to put this new knowledge in work. The goal of this chapter is the creation of a hierarchical, wire frame model of a man.

Previously acquired knowledge, like modeling and projection transformations will be used in order to create and animate this model. This chapter is divided into two sections.

In the first section, in order to discuss and explain hierarchical models and their creation, without having to cope with an overcomplicated example, instead of trying to create the whole model of the man, only its lower part will be constructed including its base, legs and feet.

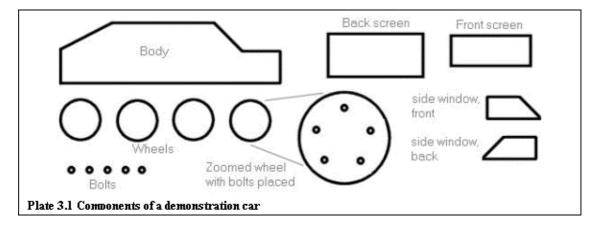
This incomplete model will then be animated. A 'walking' function will be created for this reason. When this example will be finished and enough knowledge and experience will be accumulated, the second section will follow naturally.

In the second section, the incomplete model of the man will be completed and by the end of the section a complete, three dimensional, wire–frame model of man (based on rectangles and spheres) will be ready. The 'walking' function from the previous section will be slightly modified in order to accommodate the whole body.

3.1 Building a basic hierarchical model

This section will start by describing not the code of the program but the structure of and the concepts behind hierarchical models.

Imaging that it was required to build a car for some simulation reasons. For the sake of this example this car is composed of the car's body, four wheels and six windows. There is a front window, a back window and two windows on each side of the car. The side windows are symmetrical and each wheel has five bolts (Plate 3.1).



For the purposes of this example only the right (visible in the picture) side components will be used. That is, two wheels, ten bolts and two side windows.

In order to avoid repetition it would be also be desirable to build the car in a hierarchical way; meaning that when the five bolts are correctly placed on the wheel, they should move in accordance with the wheel without any exterior help. Also it would be desirable that when the body of the car moves, its parts would not stay behind but follow its movement.

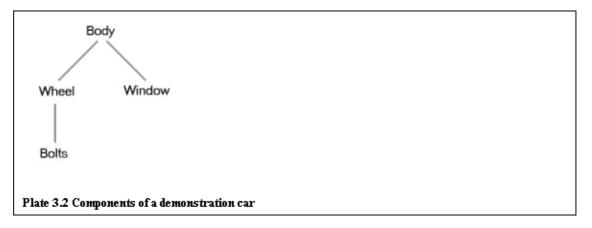
Chapter 3 - Creating a hierarchical, 3D, wire frame model

In order to achieve this a hierarchy has to be built with the car's body being the topmost item in it and then following it, the windows and the wheels, with the bolts being subordinate to the wheels. This hierarchy has the result of when the car's body is moving, the windows and wheels will move in accordance with it. Further on when the wheels rotate the bolts will rotate also.

OpenGL provides the means to build hierarchical models through the functions **glPushMatrix** and **glPopMatrix**. If it assumed that functions are available that each one of them draws a part of the car i.e. bolt, wheel, window and body then Plate 3.2 demonstrates the needed hierarchy and example 3.1 the appropriate pseudo–code to achieve it.

Example 3.1 Pseudo-code that demonstrates the car's hierarchy

```
function draw_car
{
 glPushMatrix
 draw_body_of_car
 glPushMatrix
 go_to_side_window_front_position
 draw_side_window
 go_to_side_window_back_position
 inverse_axes
 draw_side_window
                 inverse axes
 go_to_front_wheel_position
 draw wheel and bolts
 go_to_back_wheel_position
 draw_wheel_and_bolts
 glPopMatrix
      glPopMatrix
}
function draw_wheel_and_bolts
{
 glPushMatrix
 draw_wheel
 glPushMatrix
 for counter = 1 \text{ up to } 5 \text{ do}
 ł
 go_to_bolt_position
 draw_bolt
 }
 glPopMatrix
 glPopMatrix
```



In example 3.1 the function **go_to*** is used to translate to the needed point every time. The function **inverse_axes** is used to inverse the x-axis in order to use the **draw_window** function to draw the side back

window (as it is the mirror of the side front window). If now a **glTranslate** routine is issued just before the function **draw_car**, the whole car will be moved including the windows and wheels and if furthermore a relation exists that when the car moves the wheels rotate, the bolts will also rotate in accordance with wheels.

Furthermore, the commands **glPushMatrix** and **glPopMatrix** can be used to save time when positioning parts of a scene. For example lets say that the body of the car has a length of 100 units and that the co–ordinates system is positioned at the center of the car, so the car co–ordinates lie from –50 to 50. Lets also assume that the wheels have to be positioned both at ten points before the boundaries of the car. This can be done in two ways.

Without using the matrix stack, a **glTranslate**(40, 0, 0) should be issued in order to move the center of the coordinates forty units on the x-axis, then the wheel would be drawn by calling **draw_wheel_and_bolts** and then the center of the co-ordinates should be moved eighty units back in order to position the second wheel, by calling **glTranslate**(-80, 0, 0).

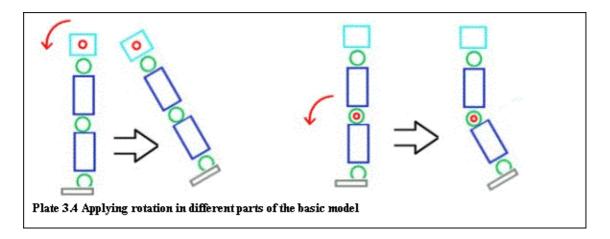
If the matrix stack is used, and the appropriate commands **glPushMatrix** and **glPopMatrix**, the same can be done in the following, more robust way. A call to **glPushMatrix** can be done in order to save the current matrix and then a call to **glTranslate**(40, 0, 0) and a call to **draw_wheel_and_bolts** can be done in order to draw the first wheel in the correct position. Then a call to **glPopMatrix** can be done in order to retrieve the prior to the translation matrix and then a call to **glTranslate**(-40, 0, 0) and a call to **draw_wheel_and_bolts** can be done in order to retrieve the prior to the translation matrix and then a call to **glTranslate**(-40, 0, 0) and a call to **draw_wheel_and_bolts** can be done in order to position and draw the second wheel.

Now that some understanding of hierarchical models and the matrix stack has been acquired, the actual design of the basic model can start. Plate 3.3 shows the parts of this first basic model and their hierarchical relation.

As it is seen in Plate 3.3, in this occasion the top most item in the hierarchy is the base of the body, followed by the upper leg joint, the upper leg, the lower leg joint, the lower leg, the foot joint and finally the foot. Joints are depicted as spheres and the other parts of the body as rectangles.

Base	Ba	se
Upper Leg	Upper Leg Joint Upper Leg Lower Leg Joint	Upper Leg Joint Upper Leg Lower Leg Joint
Lower Leg	Lower Leg	Lower Leg
Foot Joint	Foot Joint	Foot Joint
📛 📛 Foot	Foot	Foot

Chapter 3 – Creating a hierarchical, 3D, wire frame model



So now if a rotation is applied to the base, all other parts are going to be rotated whereas if a rotation is applied to the lower leg joint, only the joint and the parts lower from it will rotate (Plate 3.4).

Now is the appropriate time to go on and start constructing the code that will draw and latter on animate this model.

The project at this point is split into three files. As always the main program will residue in the file called main.c. Another file will be used called model.c that will contain all the functions that will be needed in order to position, draw and animate this model. The file model.h contains the function definitions of the file model.c.

In order to construct this basic model, some relative metrics have to be calculated that will approximate a human body. As the point of this example was not accuracy but a demonstration of hierarchical model creation, the relative heights and widths of the body parts were based on a not so accurate hand-drawn sketch of a man. Accurate modeling will be the subject of a latter chapter. Example 3.2 shows part of the file model.h, were the relative metrics of the body can be found.

Example 3.2 Size definitions of the models parts

```
#define FOOT_JOINT_SIZE HEAD_JOINT_SIZE
#define FOOT_HEIGHT FOOT_JOINT_SIZE * 2.0
#define FOOT_WIDTH LO_LEG_WIDTH
#define FOOT FOOT_WIDTH * 2.0
#define UP_ARM_HEIGHT TORSO_HEIGHT * 0.625
#define UP_ARM_WIDTH TORSO_WIDTH/4.0
#define UP_ARM_JOINT_SIZE HEAD_JOINT_SIZE * 2.0
#define LO_ARM_HEIGHT TORSO_HEIGHT * 0.5
#define LO_ARM_WIDTH UP_ARM_WIDTH
#define LO_ARM_JOINT_SIZE UP_ARM_JOINT_SIZE * 0.75
#define HAND_HEIGHT LO_ARM_HEIGHT / 2.0
#define HAND_WIDTH LO_ARM_WIDTH
#define HAND LO_ARM_WIDTH / 2.0
#define TORSO_WIDTH TORSO_HEIGHT * 0.75
#define TORSO_HEIGHT 0.8
#define TORSO TORSO_WIDTH / 3.0
#define HEAD_WIDTH HEAD_HEIGHT * 0.93
#define HEAD_HEIGHT TORSO_HEIGHT * 0.375
#define HEAD_JOINT_SIZE HEAD_HEIGHT/6
#define BASE WIDTH TORSO WIDTH
#define BASE HEIGHT TORSO HEIGHT / 4.0
#define UP_LEG_HEIGHT LO_ARM_HEIGHT
#define UP_LEG_JOINT_SIZE UP_ARM_JOINT_SIZE
#define UP_LEG_WIDTH UP_LEG_JOINT_SIZE * 2.0
#define LO_LEG_HEIGHT UP_LEG_HEIGHT
#define LO_LEG_WIDTH UP_LEG_WIDTH
```

Chapter 3 - Creating a hierarchical, 3D, wire frame model

As it seen in the example only the torso height is defined and all the other parts of the body are related to this height, for example the torso width is three quarters of the torso height etc. This was done having in mind the case that the model is needed to change dimensions, only the torso height has to be changed, as this change will affect all the other parts of the body. In this first section of the chapter not all of the previously defined parts will be needed, but nevertheless they were defined, as they will be needed in the next section, when a full body will be build.

Now that the size of the parts of the body is defined, it is the time to start building the body. The model at this point is constructed from three main parts, the 'base' (lower torso) and the two legs. Example 3.3 shows the code that creates the base.

Example 3.3 The function that draws the base of the basic model

```
void Draw_Base(int frame)
{
  glPushMatrix() ;
  glScalef(BASE_WIDTH, BASE_HEIGHT, TORSO) ;
  glColor3f(0.0,1.0,1.0) ;
  if (frame == WIRE)
  glutWireCube(1.0) ;
  else
  glutSolidCube(1.0) ;
  glPopMatrix() ;
}
```

The function **Draw_Base** accepts one argument. This argument will be used to draw either a wireframe base (by passing the value WIRE) or a solid base (by passing the value SOLID). At this point a solid base will be of no use (as the model will be wireframe) but the same function will be used later, when dealing with light, to construct a solid base.

The body of the function starts by calling the function **glPushMatrix** in order to save the current matrix before applying any modifications to it. A call to **glScale** follows with the values BASE_WIDTH, BASE_HEIGHT and TORSO. The result of this call is the scaling of the axes to these new values (from left to right, the axes x, y and z). Now when **glutWireCube**(1.0) is called (or **glutSolidCube**(1.0)) the result will not be a cube but a rectangle approximating the 'base' (as seen in Plates 3.3 and 3.4).

This function is quite easy to understand, as there is no hierarchy of objects involved or any rotations. The function **Draw_Leg** is slightly more complicated as it contains three parts, the upper leg, the lower leg and the foot. These three parts are constructed by three different functions. These three functions are similar to each other and example 3.4 shows the function that draws the upper leg, named **Draw_Upper_Leg**.

Example 3.4 The function that draws the upper leg of the basic model

```
void Draw_Upper_Leg(int frame)
{
    glPushMatrix();
    glScalef(UP_LEG_JOINT_SIZE, UP_LEG_JOINT_SIZE, UP_LEG_JOINT_SIZE);
    glColor3f(0.0,1.0,0.0);
    if (frame == WIRE)
    glutWireSphere(1.0,8,8);
    else
    glutSolidSphere(1.0,8,8);
    glPopMatrix();
    glTranslatef(0.0,- UP_LEG_HEIGHT * 0.75, 0.0);
```

Chapter 3 - Creating a hierarchical, 3D, wire frame model

```
glPushMatrix() ;
glScalef(UP_LEG_WIDTH,UP_LEG_HEIGHT,UP_LEG_WIDTH) ;
glColor3f(0.0,0.0,1.0) ;
if (frame == WIRE)
glutWireCube(1.0) ;
else
glutSolidCube(1.0) ;
glPopMatrix() ;
}
```

In the body of the function, the starting routine **glPushMatrix** is used to save the current matrix prior to the scaling. After the matrix is saved the function **glScale** is used to scale the axes in the appropriate dimensions for the drawing of the upper leg joint. When this is done, the current colour is set to green and the joint is drawn (either as wireframe or solid, depending on the value passed to the function) and then the matrix is restored by calling the function **glPopMatrix**. This has the effect of restoring the axes to their initial one–to–one relation. Following this a call to **glTranslate** is issued in order to move the centre of the axes in the new positioned required to draw the upper leg. At this point the just explained technique is repeated in order to save the matrix, scale the axes, choose the colour (blue this time) and finally draw the upper leg. At this point the function **Draw_Upper_Leg** (the function that draws the upper leg and the upper leg joint) is ready. The functions **Draw_Lower_Leg** and **Draw_Foot** are similar to this one, so they will not be explained explicitly.

Now is the time to take a look at the function **Draw_Leg**. This function combines the previously mentioned functions in order to build the whole leg, including the rotation routines, routines that will be needed for the animation of the model. Example 3.5 contains the code of this function.

Example 3.5 The function that creates the whole leg of the basic model

```
void Draw_Leg(int side, int frame)
{
  glPushMatrix();
  glRotatef(walking_angles[side][3],1.0,0.0,0.0);
  Draw_Upper_Leg(frame);
  glTranslatef(0.0,- UP_LEG_HEIGHT * 0.75,0.0);
  Jraw_Lower_Leg(frame);
  glTranslatef(0.0,- LO_LEG_HEIGHT * 0.625, 0.0);
  glRotatef(walking_angles[side][5],1.0,0.0,0.0);
  Draw_Foot(frame);
  glPopMatrix();
}
```

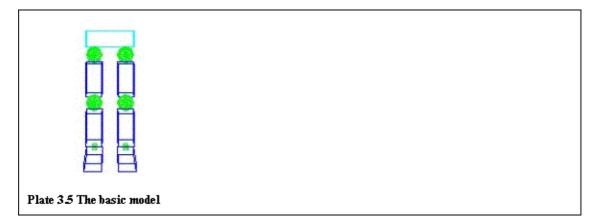
As before, the function's body starts by saving the current matrix. Next is a call to **glRotate**. The values passed to this routine show that the object that is drawn after this function is called, will be rotated only on the x-axis (as the second parameter is 1.0 and the third and fourth are 0.0). The first parameter, is the amount of degrees the x-axis should be rotated. This value is contained in the array *walking_angles*. This is a two dimensional array of size two by six, that is declares in the file main.c (and is available to this file by declaring it as **#extern**) and contains all the required, for the walking animation, angles. Its structure is such that will keep six rotation angles (upper arm, lower arm, hand, upper leg, lower leg and foot) for both sides (left and right arms and legs).

Following that, the function **Draw_Upper_Leg** is called in order to draw the upper part of the leg. Next the centre of the axes is moved to the new required position by calling the routine **glTranslate** and the rest of the function continue to a similar to the just described manner (rotate axes, draw part and move the centre of the axes to the new position). When the leg is created (including upper leg, lower leg and foot) the function **glPopMatrix** is used to restore the initial (prior to this function) matrix.

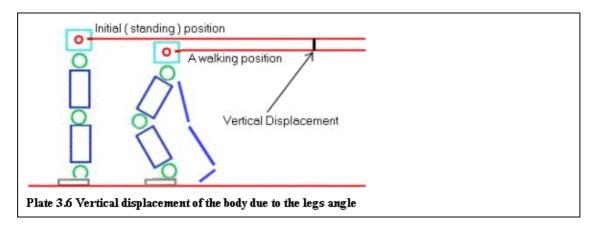
Now that two functions are ready, one that draws a base and one that draws a leg, it is quite straight forward what is needed in order to have the completed (for this section) model. Example 3.6 shows the code needed in order to build finally the basic model.

Example 3.6 The function that creates the basic model

```
void Draw_Base_Legs(void)
{
  glPushMatrix() ;
  glTranslatef(0.0,base_move,0.0) ;
  Draw_Base(WIRE) ;
  glTranslatef(0.0,-(BASE_HEIGHT),0.0) ;
  glPushMatrix() ;
  glTranslatef(TORSO_WIDTH * 0.33,0.0,0.0) ;
  Draw_Leg(LEFT,WIRE) ;
  glPopMatrix() ;
  glTranslatef(-TORSO_WIDTH * 0.33,0.0,0.0) ;
  Draw_Leg(RIGHT,WIRE) ;
  glPopMatrix() ;
}
```



As it is seen in example 3.6, just after saving the current matrix by calling the routine **glPushMatrix** a call to the routine **glTranslate** is done with one of its parameters being the value *base_move*. The particular call will be explained in a while. Following that, the base is drawn by calling the function **Draw_Base**. Next the centre of the axes is moved lower in order to draw the legs. The matrix is saved, the axes are moved to the left and the left leg is drawn; the matrix is restored, the axes are moved to the right and the right leg is drawn. Finally the routine **glPopMatrix** is called in order to restore the initial matrix. If this program is compiled and run the results will be the ones shown in Plate 3.5



In example 3.6 the first call to the routine **glTranslate** was left without an explanation. As it can be seen a value is passed to this routine, named *base_move*. This value is the vertical displacement of the body, due to

the walking animation. When a human walks, its torso does not remain at the same point but moves slightly up and down due to the angle of the legs (Plate 3.6). Example 3.7 contains the function that calculates this vertical displacement.

Example 3.7 The function that calculates the vertical displacement of the body

```
double find_base_move(double langle_up, double langle_lo, double rangle_up, double rangle_lo)
double result1, result2, first_result, second_result, radians_up, radians_lo ;
radians_up = (PI*langle_up)/180.0 ;
 radians_lo = (PI*langle_lo-langle_up)/180.0 ;
 result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up) ;
 result2 = (LO_LEG_HEIGHT + 2 * (LO_LEG_JOINT_SIZE + FOOT_JOINT_SIZE) + FOOT_HEIGHT)
            * cos(radians_lo) ;
 first_result = LEG_HEIGHT - (result1 + result2) ;
radians_up = (PI*rangle_up)/180.0 ;
radians_lo = (PI*rangle_lo-rangle_up)/180.0 ;
result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up) ;
 result2 = (LO_LEG_HEIGHT + 2 * (LO_LEG_JOINT_SIZE + FOOT_JOINT_SIZE) + FOOT_HEIGHT)
            * cos(radians_lo) ;
 second_result = LEG_HEIGHT - (result1 + result2) ;
 if (first_result <= second_result)</pre>
return (- first_result) ;
else
return (- second_result) ;
}
```

As it can be seen in Plate 3.7 the vertical displacement *VD* can be calculated by subtracting the values *upper_leg_vertical* and *lower_leg_vertical* by the leg's length, LL:

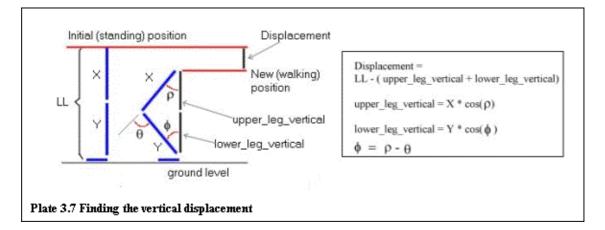
VD = LL – (upper_leg_vertical + lower_leg_vertical) (1)

At this point the vertical displacement due to the foot is not taken into account.

Back to the function **find_base_move**, the angles are firstly converted from degrees to radians (as the library routine **cos** that is used to find the cosine of the angles needs the angles to be in radians). Then the previously defined function (1) is used to find the vertical displacement. In the function, *VD* is represented as *final_result*, *upper_leg_vertical* as *result1* and *lower_leg_vertical* as *result2*. To find *result1* and *result2* the following functions are used (consult Plate 3.7):

$$result1 = X * cos(r) (2)$$

$$result2 = Y * cos(f) (3)$$



Chapter 3 - Creating a hierarchical, 3D, wire frame model

The vertical displacement for both legs is found and then a check is done to see which one of the two is touching the ground (its vertical displacement will be less than the others will); this value is then returned by the function.

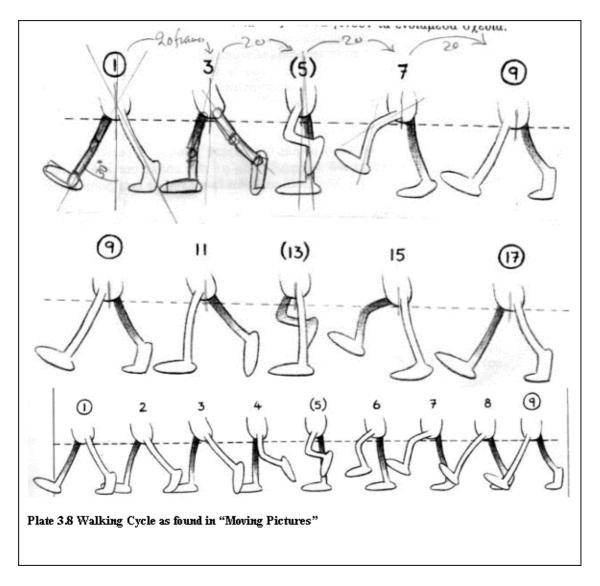
At this point there is available to the user a function that draws a basic model; there is also a function that is able to calculate the vertical displacement of this particular model. A remaining function to construct, is a function that will give life to this model, an animation function that will make the model walk.

In this, first section, of the chapter the angles of the walking animation will be 'hardwired', meaning that the program will not read them from a file but they will exist in the body of the animation function. Later, in the second section of this chapter, this will change, as the program will become data driven (it will read all its data from files).

This function will be based on the technique of key framing. This technique firstly identifies a number of key frames. These key frames are frames where something important for the animation happens. At these key frames the angle of every part of the body will be provided to the program, meaning that the programmer will explicitly calculate and pass these angles to the function. Then the function will use these key frames to calculate the angles of every part of the body for every frame of the animation.

This will be accomplished by taking the angle between two key frames and divide this angle among the other frames. For example, if the lower part of the leg has to be moved twenty degrees between two key frames and this has to be done in twenty frames, the function will calculate that the lower part of the leg has to be moved one degree every single frame (twenty degrees divided by twenty frames = one degree per frame), in order to accomplish the stated need.

The walking animation function was based on the book by Tony Wight "Moving Pictures". In this book a walking animation cycle was provided based on eight key frames. The first four key frames were used in order to animate the first half of the walking movement and the rest four in order to animate the second half. The second half of the animation is the same as the first part but in reverse. In the first half of the animation the leg that was in front before the animation starts will end up being behind and the leg that was behind will end up in front. The second half of the animation does just the reverse of first half of the animation in order to complete the walking cycle and start from the beginning for a new cycle.



The animation function that was build for the previously discussed model, is based on the sketches found in this book, so its structure follows the structure that was described in the previous paragraph. The angles used were calculated from the sketches in the book. Plate 3.8 shows the walking cycle as appeared in the book. Example 3.8 contains part of the code of this function.

Example 3.8 Part of the walking animation function

```
void animate_base(void)
{
 static frames = FRAMES,
 zoom_fl = 0,
 flag = 1 ;
 float l_upleg_dif ,
 r_upleg_dif ,
 l_upleg_add ,
 r_upleg_add ,
 l_loleg_dif ,
 r_loleg_dif ,
 l_loleg_add ,
 r_loleg_add ;
 switch (flag)
 {
 case 1 :
 l_upleg_dif = 15 ;
 r_upleg_dif = 5 ;
```

```
l_loleg_dif = 15 ;
r_loleg_dif = 5 ;
l_upleg_add = l_upleg_dif / FRAMES ;
r_upleg_add = r_upleg_dif / FRAMES ;
l_loleg_add = l_loleg_dif / FRAMES ;
r_loleg_add = r_loleg_dif / FRAMES ;
walking_angles[0][3] += r_upleg_add ;
walking_angles[1][3] += l_upleg_add ;
walking_angles[0][4] += r_loleg_add ;
walking_angles[1][4] += l_loleg_add ;
langle_count -= l_upleg_add ;
langle_count2 -= l_loleg_add ;
rangle_count -= r_upleg_add ;
rangle_count2 -= r_loleg_add ;
base_move = find_base_move ( langle_count, langle_count2, rangle_count, rangle_count2 ) ;
frames-- ;
if (frames == 0)
flag = 2;
frames = FRAMES ;
 }
break ;
case 2 :
     ...... repeat until case 8 then go to case 1......
if (zoom_flag)
 {
 switch (zoom_fl)
 {
 case 0 :
 zoom += 0.05 ;
 if (zoom > 2.5) zoom_fl = 1 ;
 break ;
 case 1 :
 zoom -= 0.05 ;
 if (zoom < -2.5) zoom_fl = 0;
 break ;
 default :
 break ;
 }
}
if (rotate_flag)
{
rotate = (rotate + 1) % 360 ;
}
glutPostRedisplay() ;
}
```

At the start of this function some variables are declared. The variables *frames*, *zoom_fl* and *flag* are declared as **static** because they are needed to be initialised only once (the first time the function is called). Just after the variables declaration a **switch** statement follows. This is the skeleton of the function, as all the operations needed to be done for the walking animation happen inside this statement.

This **switch** statement depends on the variable *flag*, which is initially set to 1. This means that the first part of this statement (the one under the label 'case 1 :') will be executed until the variable *flag* changes from 1 to a different value (in this case it will eventually become 2).

Inside this part of the **switch** statement, the variables l_upleg_dif, l_loleg_dif, r_up_leg_dif and r_loleg_dif are initialised to some values. These values are the difference of the angles of the left and right upper and lower leg between the first two key frames.

After this the variables l_upleg_add, l_loleg_add, r_upleg_add and r_loleg_add are calculated, by dividing the initial angle difference (between the two key frames) by the number of frames that are needed in order to make the animation. These will be the rotation values for a single frame animation. The number of needed frames between two key frames is constant and is defined in this case as twenty in the file model.h.

The next step is to copy the values of the previously calculated variables in the proper places in the array *walking_angles*. This, externally defined array that was described previously is used in the **draw_base** function in order to animate the model. The values of the variables are not just copied but they are added to the previous value of the array in order that the array will contain the angles for the next frame. This is done because the rotation is not incremental; for example if the function **glRotate**(20, 1.0, 0.0, 0.0) was used to rotate the upper leg by twenty degrees and at the next step the upper leg is needed to be rotated by five degrees more, the correct call will be **glRotate**(25, 1.0, 0.0, 0.0) and not **glRotate**(5, 1.0, 0.0, 0.0). *OpenGL* follows this non–incremental technique in order to diminish cumulative errors that may appear if this particular modelling transformation was based on an incremental technique.

After this is done these values are subtracted from the variables langle_count, langle_count2, rangle_count and rangle_count2. These four variables are initialised externally, in the file main.c, and contain the initial values of the angles of the body parts. These variables will be used with the function **base_move** in order to calculate the body's vertical displacement. By doing so the values of the variables r (*angle_count) and q (*angle_count2) of both left and right legs (1 / r) are retrieved (review Plate 3.7).

After the new angles are calculated by the technique explained in the previous paragraph, the values of these variables are passed to the function **base_move**, in order to find the vertical displacement of the body.

This is the end of the first cycle (transition from key frame one to key frame two). The variable *frames* is decrement and a check is done to see if the value of *frames* is equal to 0. If it is 0, it means that the second key frame is reached and that the value of the variable *flag* must be incremented (in order to move to the next **case** in the **switch**, the second cycle). The variable *frames* is also reinitialised to FRAMES (the defined, constant number of frames between two key frames).

This will continue until the end of **case** 8 will be reached and then *flag* will be set to 1 for the walking cycle to start from the beginning. At the end of the switch statement some more code is visible in example 3.8. This code calculates the value of the variables *zoom* and *rotate*. These variables are used externally, in the main program to zoom in and out (on the z-axis) and rotate the model (on the y-axis).

	Keys and their operation
S	Single frame animation
S	Continues animation
đ	Start/Stop Rotation of the model
D	Start/Stop zooming in and

Now that all the main functions of the program are ready, only one is left in order to finish the program. This is the **keyboard** function that will provide the needed interaction between the user and the program. This function has the same structure as the one described in example 2.11, so it will not be examined here. For reference, table 3.1 contains the keys that are used by the program and their operations.

At this point the program is nearly ready, as only a couple of operations remain to be done in the main program in order to have a fully working program. In the main program all the previously described as external variables are declared. When this is done, the variables langle_count, langle_count2, rangle_count and rangle_count2 are initialised to the values 30, 0, -30 and 0. These, as described before, are the initial

angles of both left and right, upper and lower leg. The variables zoom_flag and rotate_flag are initialised to GL_FALSE (at first the model will not be zoomed or rotated) and the variables rotate and zoom are set to 0.0, as the model is initially not zoomed nor rotated.

Something new appears also in the function init. Example 3.9 contains the code of this function.

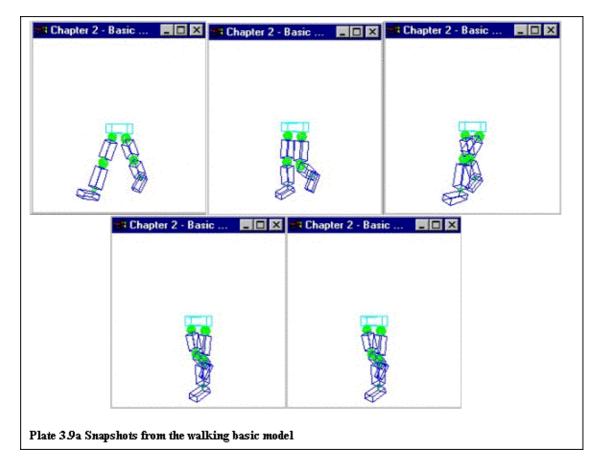
Example 3.9 The init function that prints out general information about the OpenGL version

```
void init(void)
ł
const GLubyte* information ;
glClearColor(1.0, 1.0, 1.0, 0.0);
glShadeModel(GL_FLAT) ;
 information = glGetString(GL_VENDOR) ;
printf("VENDOR : %s\n", information) ;
 information = glGetString(GL_RENDERER) ;
printf("RENDERER : %s\n", information) ;
 information = glGetString(GL_EXTENSIONS) ;
printf("EXTENSIONS : %s\n", information) ;
 information = glGetString(GL_VERSION) ;
printf("VERSION : %s\n", information) ;
walking_angles[0][3] = langle_count ;
walking angles[1][3] = rangle count ;
walking_angles[0][4] = langle_count2 ;
walking_angles[1][4] = rangle_count2 ;
base_move = find_base_move( langle_count, langle_count2, rangle_count, rangle_count) ;
}
```

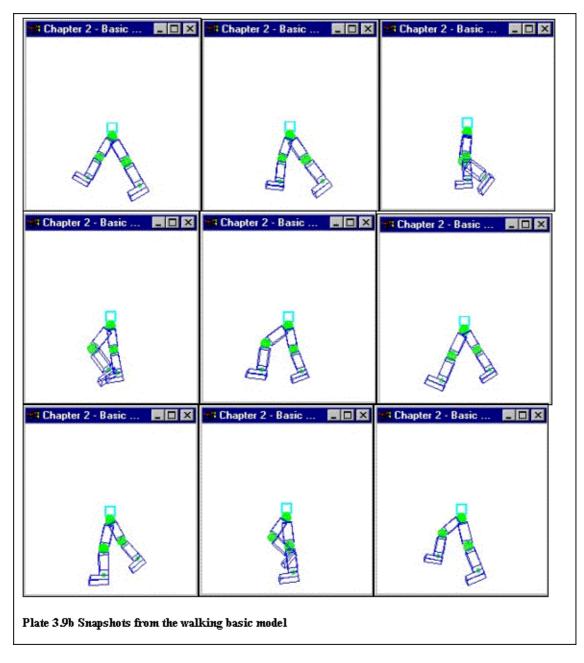
In this function the variable information (of type Glubyte pointer) is used with the *OpenGL* function **glGetString** in order to retrieve and then print general information about the *OpenGL* version, vendor, extensions supported, etc.

In this function the array that is used to store the angles is also initialised. The initial vertical displacement of the model is found also by calling the function **base_move** and passing the legs initial angles.

A new callback function is also used in this program. The function **glutSpecialFunc** is similar to the **glutKeyboardFunc** but is used to register a callback function responsible for the keys that do not generate an ASCII code, like the directional keys, the Control and Alt key, and the Function keys (F1 to F12). The structure of this function is similar to the one of the keyboard function shown in example 2.11. After registering the function **special** by calling the function **glutSpecialFunc** in the **main** function the user will be able to use the up and down directional keys to zoom in and out of the model and the left and right directional keys to rotate the model on the y-axis. The calls to the routines **glTranslatef** (0.0, 0.0, zoom) and **glRotatef** (rotate, 0.0, 1.0, 0.0) just before drawing the model in the **display** function will allow for the zooming and rotation effects.



At this point, the first section of the second chapter is completed. After compiling and run the program the user will be able to see this basic model walking. Plate 3.9 contains some screen–shots that were taken from this program.



3.2 Improving the basic model

The goal of this section is to create a wireframe model of a man that will be able of walking. This program will be based on the previously constructed (in section 3.1) program. The structure of this new program will be similar to the previous one with the difference that this second program will have a more 'professional' touch. For example the animation angles will be read from a file instead of being hardwired in the walking procedure; the program will also be split into five different parts that each one of them will contain relevant functions. These five files are going to be main_model.c, model.c, inout.c, anim.c and keyboard.c. Each one of them will also have its header file. Another file will also be constructed that will contain general definitions, general.h.

This section's keyboard interaction is the same as in the previous section, so no particular interest will be given to it. The animation function, **animate_body**, is also similar to the one used in the previous program, the only difference being that now, instead of calculating the walking angles of the legs, it calculates the walking angles of the arms also. The technique used to calculate the arms walking angles is the same as the previously explained one (the one that is used to calculate the legs walking angles), so there is no reason for explicit demonstration of this new function.

Chapter 3 - Creating a hierarchical, 3D, wire frame model

The file inout.c contains the newly created functions that are responsible for file input–output (reading the angles from a file and other file related functions). The custom function **Open_Files** was created for this particular program and accepts one argument. Depending on the argument it can open two different files. If the argument is 'r' it will try to open the file 'data.txt', for reading, from three pre–defined directories, 'e:\bp\chapter3\model2\', 'g:\data\', or 'a:\', in order to read the walking animation angles. If the file is not found in this directories the function notifies the user that the file was not found and the program exits. If the argument is 'w' the function will try to open the file 'test.txt' for writing in the same three directories. This function will be improved in a later example in order to check for the files, not in previously defined directories but in the directory the actual program is found. The file 'test.txt' is used later on from a function in order to print the walking angles for testing reasons.

The next function implemented in this file will use the structure *anim_angles* to store the animation angles read from the file. This structure is defined in the file general.h and can be found in example 3.10.

Example 3.10 The definition of the structure *anim_angles*

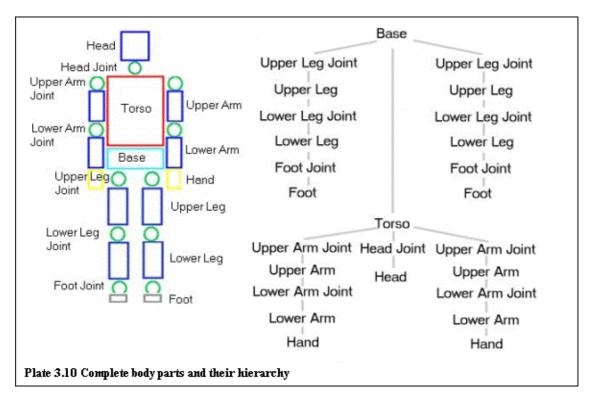
```
typedef struct
ł
float head ;
float upbody ;
 float lobody ;
 float l_uparm ;
 float l_loarm ;
 float l_hand ;
float l_upleg ;
 float l_loleg ;
float l_foot ;
float r_uparm ;
float r_loarm ;
 float r_hand ;
 float r_upleg ;
 float r_loleg ;
 float r_foot ;
} anim_angles ;
```

As it can be seen in this example the structure $anim_angles$ contains fifteen elements of type *float*. Each one of these elements will store the animation angle for a particular body part, for example the left upper $arm(l_uparm)$ in the structure), etc.

The function **Read_Data_From_File** calls the previously described function **Open_Files**('r') in order to open the file data.txt for reading. This function accepts two arguments of type *anim_angles*. The first one named *init* is actually a pointer to the particular structure and is used to store the initial (prior to the animation) angles. The second argument, *array*[], is an array of four *anim_angles* elements. In this array the angles of the first four key frames will be stored. As the key frames are symmetrical (the last four to the first four) the values of this array will also be used to find the angles of the last four keyframes. The structure of this function is very simple as it just uses **fscanf** calls to read the angles from the file and place them in one of the two just mentioned variables (*init* or *array*). An integer variable named *scan_counter* is also used in this function to count how many values are actually read from the file. The number of the angles read is then output to the screen for testing reasons.

The last function implemented in this file is the function **Write_Test_Data**. This function accepts the same two arguments, the function **Read_Data_From_File** had, but instead of initialising them it uses them to output their values into a file, for testing reasons. Its structure is very simple, as it just calls the function **Open_Files**('w') to open the file 'test.txt' for writing and then with several **fprintf** calls, it writes the angles of the animation in the file. By comparing the two files, 'data.txt' and 'test.txt' a user can find out if the program reads in the correct animation angles.

Now is the time to take a look at the contents of the file model.c. This file contains all the functions that are responsible for drawing the model on the screen. It contains all the previously implemented model functions like **Draw_Upper_Leg**, **Draw_Lower_Leg**, **Draw_Foot**, **Draw_Leg**, etc.



It also contains the newly created functions that draw the head, the upper arm, the lower arm, the hand and the torso. These functions were constructed in a manner similar to the one described in the first section of the chapter. A function that draws all the parts together, in order to draw the complete model of a man, was also created and implemented in this file. This pseudocode of this function, named **Draw_Model** can be seen in example 3.11. Plate 3.10 contains the parts that constitute the new improved model and their hierarchy.

Example 3.11 The pseudo-code of the function that creates and draws the complete model of a man.

```
function Draw_Model
{
save_the_matrix (prior to this function)
create_base
save_the_matrix (to place the second in the hierarchy torso)
translate_to_correct_place
 create_torso
 save_the_matrix(to place the third in the hierarchy head)
 translate_to_correct_place
create_head
restore_the_matrix
restore_the_matrix
 save_the_matrix (to place the second in the hierarchy arms)
 translate_to_correct_place
 create_left_arm
 translate_to_correct_place
 create_right_arm
restore_the_matrix
 save_the_matrix (to place the second in the hierarchy legs)
 translate_to_correct_place
create_left_leg
 translate_to_correct_place
create_right_leg
restore_the_matrix
```

Chapter 3 - Creating a hierarchical, 3D, wire frame model

```
restore_the_matrix
}
```

The structure of this function is similar to the structure of the function **Draw_Base_Legs**, which was described in the first section of this chapter.

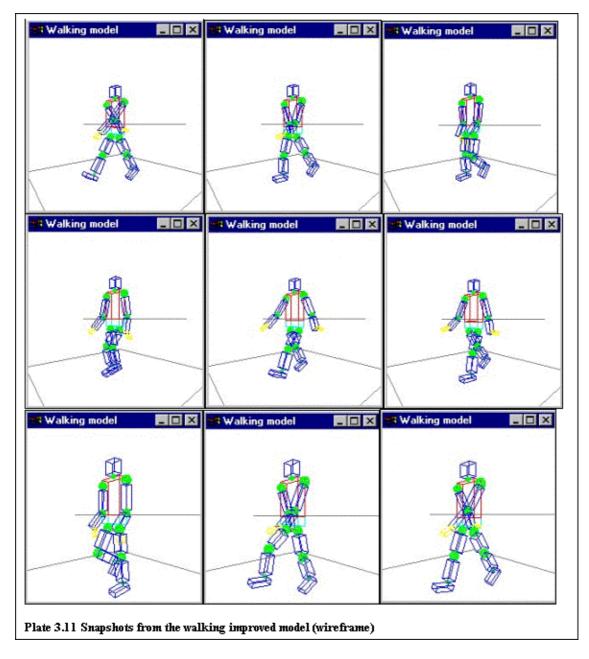
Firstly the matrix prior to this function is saved, then the base is created and the matrix is saved again (to place the second in the hierarchy) torso. The centre of the co–ordinates is moved to the correct place and the torso is created. The matrix is saved again (to place the third in the hierarchy) head, the co–ordinates are moved to the new place and the head is created. The matrix is restored twice (to climb up the hierarchy twice) and the just explained technique is repeated in order to create the legs and arms. Example 3.12 contains the code of the function **Draw_Model**, based on the pseudo–code shown in example 3.11.

Example 3.12 The function that creates and draws the complete model of a man.

```
void Draw_Model(int frame)
{
glPushMatrix() ;
glTranslatef(0.0,base_move,0.0) ;
Draw_Base(frame) ;
glPushMatrix() ;
glPushMatrix() ;
glTranslatef(0.0, TORSO_HEIGHT / 2.0, 0.0) ;
Draw_Torso(frame) ;
glPopMatrix() ;
glPushMatrix() ;
glPushMatrix() ;
glTranslatef(0.0, TORSO_HEIGHT + (HEAD_HEIGHT/2.0) +HEAD_JOINT_SIZE * 2.0, 0.0) ;
Draw Head(frame) ;
glPopMatrix() ;
glPopMatrix() ;
glPushMatrix() ;
glTranslatef(0.0,TORSO_HEIGHT * 0.875,0.0) ;
glPushMatrix() ;
glTranslatef(TORSO_WIDTH * 0.66, 0.0,0.0) ;
Draw_Arm(LEFT,frame) ;
glPopMatrix() ;
glTranslatef(- (TORSO_WIDTH * 0.66), 0.0,0.0) ;
Draw_Arm(RIGHT,frame) ;
glPopMatrix() ;
glPushMatrix() ;
glTranslatef(0.0,-(BASE_HEIGHT*1.5),0.0) ;
glPushMatrix() ;
glTranslatef(TORSO_WIDTH * 0.33,0.0,0.0) ;
Draw_Leg(LEFT, frame) ;
glPopMatrix() ;
glTranslatef(-TORSO_WIDTH * 0.33,0.0,0.0) ;
Draw_Leg(RIGHT, frame) ;
glPopMatrix() ;
glPopMatrix() ;
}
```

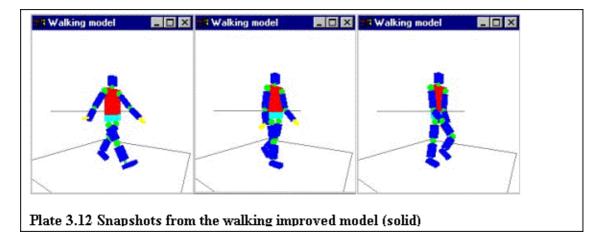
Now that the functions that draw and animate the body are ready, only a couple of steps remain before having a complete and ready to run program.

The file main_model.c is similar to the first sections, main.c file. The only differences being the declaration of the variable *init_angles* and the array *angles*[4], that will be used from the in–out functions to store the animation angles.



A second difference is found in the function **display**, where instead of the call **Draw_Base_Legs** that was used in the previous section to draw the incomplete model, a call to **Draw_Model** is done to draw the complete model. Prior to drawing the model some code can be found. This code (shown in example 3.13) is responsible for creating a wireframe rectangle (at the place of where the floor should be) and a horizontal line near the 'base' of the model. These two serve as reference for the user, in order to help him see the vertical displacement of the body.

Chapter 3 – Creating a hierarchical, 3D, wire frame model



At this point the program is ready. If it is compiled and run, the results can be found in Plate 3.11. If instead of using the value WIRE when calling the function **Draw_Model** the value SOLID is used, the results can be found in Plate 3.12. The solid model (shown in Plate 3.12) will be much improved in the next chapter with the addition of light.

Chapter 4– Lighting

As already demonstrated, OpenGL computes the colour of each pixel in a scene and that information is held into the frame buffer. Part of this computation depends on what lighting conditions exist in the scene and in which way objects absorb and/or reflect light. Actually in some cases the objects appear invisible until light is added.

Light is very important in real life as well as in graphics. For example, the sea looks bright green in the morning, blue during the day and black during the night. The colour of the water does not actually change, as it is always transparent, but the reflection conditions change.

By using OpenGL, the lighting conditions and the properties of the objects can be changed in order to produce many, sometimes stunning effects.

OpenGL approximates light and lighting as if light could be broken into red, green and blue components (R–G–B colour model). Thus, the colour of the light sources can be characterised from the amounts of the red, green and blue light they emit, and the material of an object characterised by the percentage of red, green and blue light it reflects in various directions.

In the OpenGL lighting model, light comes from several light sources in the scene that can be individually turned on and off. Some light comes from a particular direction and some light is scattered around the scene. For example, when you turn on a light bulb in a room, some light arrives at a particular object in the room directly from the bulb and some light arrives at the object after bouncing on one or more other surfaces, like walls, furniture, etc. This bounced light is called ambient and it is assumed that it is so scattered that it does not come from any particular direction, but from everywhere.

In the OpenGL lighting model, a light source has an effect only when some surface that absorbs and/or reflects light is present. Each surface is composed of a material that has various properties. A material might emit its own light (like headlights in cars), might absorb some percentage of the incoming light and might reflects some light in a particular direction.

The OpenGL lighting model considers light to be divided into four independent components: emissive, ambient, diffuse and specular. All four components are computed independently and then added together.

A fifth element might influence the appearance of an object and that is the shininess of the object. Depending on the shininess, a particular object reflects the incoming specular light in different ways.

This chapter is divided into five sections; each one dedicated to a particular lighting effect.

The first example draws four wireframe and four solid cubes. Some of them are drawn with lighting turned on, while some of them are drawn with lighting turned off in order to demonstrate the difference in appearance of objects when lighting is used.

In the second example, the OpenGL feature colour tracking is demonstrated. Normally when lighting is enabled, the objects must have a material assigned to them. Depending to the material used objects appear deferent when lit (like in the real world). For example a sphere that has a 'wooden' material (ambient, diffuse, specular and emissive values that approximate wood's behaviour) will look different from a 'silver' sphere. A material (in OpenGL) is what colour is to programs that do not use lighting. By using the colour tracking feature a programmer might choose to assign colours to objects (instead of materials) and OpenGL will convert them to materials. In some cases this is quite useful, as it is much simpler to assign colour from assigning materials.

Assigning materials and manipulating their properties is the topic of the third example. Three red cubes are created and different values of shininess are assigned to each of them in order to demonstrate this particular

light property.

In the fourth section of the chapter a program called 'Material–Lights' will be created. A user will be able to change the material and light properties of several objects interactively in order to become familiar with the concept. The program will also be able to save a particular 'colour' (a combination of material and light conditions) for latter reference. The concept of windows and sub–windows will also be discussed in this section.

Finally, the goal of the fifth section of this chapter will be to improve the previously constructed model of a man. The model that was created in the last section of the previous chapter will be taken and its structure will be slightly modified in order to become a solid, lighted model.

4.1 Getting started with lighting

In this first section of this chapter, the necessary steps to create a light source will be explained. When lighting is used with objects that are not supposed to be drawn using lighting some strange effects appear. In order to demonstrate these effects and the correct use of lighting, this example draws four wireframe cubes and four solid cubes, each one of them with different lighting conditions. These lighting conditions will be a combination of the following: lighting enabled, lighting disabled, depth testing enabled and depth testing disabled. Depth testing is an OpenGL feature that does hidden surface removal by using the depth buffer.

When drawing solid, lighted objects it is very important to draw the objects that are nearer to the viewing position and eliminate any objects obscured by others nearer to the eye.

The elimination of parts of solid objects that are obscured by others is called hidden–surface removal. The easiest way of achieving this in OpenGL is to use the depth buffer. In order to use the depth buffer, a window must be created that will have such a buffer. Passing the argument GLUT_DEPTH in the function **glutInitDisplayMode** does this. When this is done the OpenGL function **glEnable** can be called with the value GL_DEPTH_TEST in order to add hidden–surface removal to the particular program.

This program is based on the example in the fourth section of chapter 2. It uses all the functions that were defined there in order to draw and position the eight cubes (four wireframe and four solid ones).

A difference is that all the cubes are drawn by using perceptive projection and that **glEnable** and **glDisable** statements appear inside the display function in order to activate and deactivate lighting and hidden–surface removal.

Before using lighting, at least one of OpenGL's lights must be enabled. For this example one light is enough, and passing the value GL_LIGHT0 to the routine **glEnable** (inside the body of the **init** function) has the effect of activating one light. Different OpenGL implementations may provide different amounts of lights but all of them have at least eight lights. Example 4.1 contains the display function of this program.

Example 4.1 Display function that draws eight cubes with/without lighting and depth testing

```
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ) ;
   glViewport(0,win_size_V / 2, win_size_H / 4 ,win_size_V / 2) ;
   glDisable(GL_LIGHTING) ;
   glDisable(GL_DEPTH_TEST) ;
   Draw_Cube_Transl_Rot(WIRE) ;
```

}

```
glViewport(win_size_H / 4,win_size_V / 2, win_size_H / 4 ,win_size_V / 2) ;
glEnable(GL_DEPTH_TEST) ;
Draw_Cube_Transl_Rot(WIRE) ;
glViewport(2 * (win_size_H / 4),win_size_V / 2, win_size_H / 4 , win_size_V / 2) ;
glEnable(GL_LIGHTING) ;
glDisable(GL_DEPTH_TEST) ;
Draw_Cube_Transl_Rot(WIRE) ;
glViewport(3 * (win_size_H / 4),win_size_V / 2, win_size_H / 4 , win_size_V / 2) ;
glEnable(GL_DEPTH_TEST) ;
Draw_Cube_Transl_Rot(WIRE) ;
glViewport(0,0, win_size_H / 4 ,win_size_V / 2) ;
glDisable(GL_LIGHTING) ;
glDisable(GL_DEPTH) ;
Draw_Cube_Transl_Rot(SOLID) ;
glViewport(win_size_H / 4,0, win_size_H / 4 ,win_size_V / 2) ;
glEnable(GL_DEPTH_TEST) ;
Draw_Cube_Transl_Rot(SOLID) ;
glViewport(2*(win_size_H / 4),0, win_size_H / 4 ,win_size_V / 2);
glDisable(GL_DEPTH_TEST) ;
glEnable(GL_LIGHTING) ;
Draw_Cube_Transl_Rot(SOLID) ;
glViewport(3*(win_size_H / 4),0, win_size_H / 4 ,win_size_V / 2);
glEnable(GL_DEPTH_TEST) ;
Draw_Cube_Transl_Rot(SOLID) ;
glutSwapBuffers() ;
```

As it seen in this example, the **display** function is divided into eight similar parts. Each one of them calls the routine **glViewport** in order to specify where the particular cube should be drawn. The first four cubes (upper part of the window) are drawn as wireframes, while the last four are drawn as solid (lower part of the window).

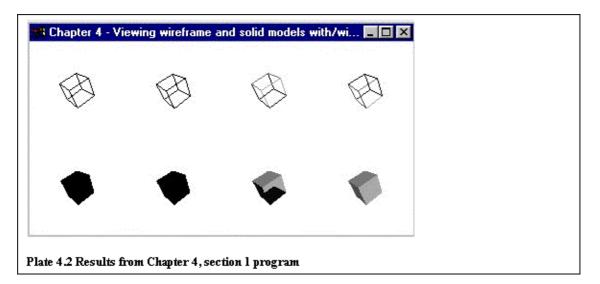


Plate 4.2 contains the results of the compiled and executed program. The remaining parts of the program are not discussed, as they are the same ones used in Chapter 2, section 4.

As seen in Plate 4.2 the best looking wire frame cube is the top, left-most cube and the best looking solid

cube is the bottom right-most one.

From this example some observations may be made. When drawing wire frame objects, depth testing does not have any effects (as it does hidden–surface removal and not hidden–line removal). Also when drawing wire frame objects all lights should be disabled, as in the opposite case the objects do not appear clear (i.e. the top, two cubes on the right of Plate 4.2).

On the other hand if the lower part of Plate 4.1 is observed, it can be seen that when solid models are drawn, lighting should be enabled, otherwise the objects do not appear three–dimensional. Depth testing should also be enabled when drawing solid, lighted models as in the opposite case (when depth–testing is disabled), the different parts of the object may be drawn in the wrong order with the results shown in the lower part of Plate 4.2, second cube from the right hand–side.

This happens because when depth testing is not enabled, no information is held about the depth of the objects on the screen relative to the viewpoint, so no calculation can be done in order to hide surfaces that are not visible.

4.2 Colour Tracking

As mentioned in the introduction of this chapter, colour tracking is an OpenGL feature that enables the programmer to assign colours instead of materials to objects that are going to be used in programs that use lighting. Colour tracking minimises also performance costs associated with material assigning.

This is a very useful feature of OpenGL, as it removes the overhead of having to assign manually the material properties of objects when something like that is not needed. If, for example a program just needs a simple red sphere and the properties of the material are of no importance (i.e. just a red sphere not a 'wooden' or 'metal' red sphere), the routine **glColor** can be used in conjunction with colour tracking in order to achieve the same effect more easily.

In order to demonstrate what colour tracking does, three solid cubes will be drawn on the screen each one having a different colour assigned to it (red, green and blue) with the routine **glColor**. Example 4.2 contains the code of the particular display function.

Example 4.2 Display function that draws three cubes (a red, a green and a blue one)

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ) ;
    glShadeModel(GL_SMOOTH) ;
    glViewport(0,0, win_size_H / 3 ,win_size_V) ;
    glColor3f(1.0,0.0,0.0) ;
    Draw_Solid_Cube_2() ;
    glViewport(win_size_H / 3,0, win_size_H / 3 ,win_size_V) ;
    glColor3f(0.0,1.0,0.0) ;
    Draw_Solid_Cube_2() ;
    glViewport(2*(win_size_H / 3),0, win_size_H / 3 ,win_size_V) ;
    glColor3f(0.0,0.0,1.0) ;
    Draw_Solid_Cube_2() ;
    glViewport(2*(win_size_H / 3),0, win_size_H / 3 ,win_size_V) ;
    glColor3f(0.0,0.0,1.0) ;
    Draw_Solid_Cube_2() ;
    glutSwapBuffers() ;
}
```

In this code, as it can be seen three viewports are defined, one for each.. Their colours are (from left to right)

red, green, and blue. The cubes are drawn by using the previously defined function **Draw_Cube_Transl_Rot** (Second Chapter). This function is slightly modified, in order to set the cube's colour outside the function.

If this example is compiled and run, with lighting enabled (and the basic LIGHT0), the results will be the ones demonstrated in Plate 4.3.

The three cubes appear grey scaled and not colour because colour tracking was not enabled. As the material of the cubes was not specified, but instead calls to **glColor** were used, colour tracking must be enabled in order for the cubes to appear in colour.

This can be done either in the **display**, or in the **init** function by calling the routine **glColorMaterial**. This function accepts two arguments, the first one being the polygon face that colour tracking is to be enabled and the second is one of the four light components (diffuse, specular, ambient and emissive).

The polygon face can be the back face (GL_BACK), the front face (GL_FRONT) or both the back and front face (GL_FRONT_AND_BACK). By default front-facing polygons are the polygons whose vertices appear in a counter-clockwise order on the screen. Using the function **glFrontFace**, and supplying the desired front-face orientation (either GL_CCW for counter-clockwise orientation or GL_CW for clockwise orientation) can change what appears to be front-facing polygons.

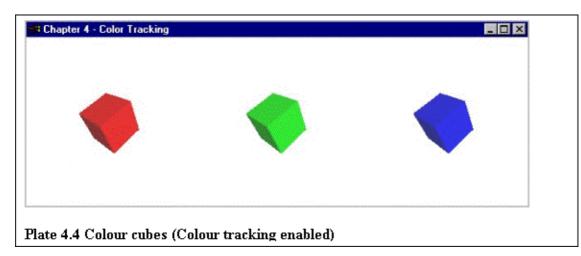


Plate 4.4 contains the results of the program if colour tracking is enabled with the parameters GL_FRONT and GL_DIFFUSE.

In order to use colour tracking the function **glEnable** must be called with the parameter GL_COLOR_MATERIAL, just after calling the function **glColorMaterial**.

4.3 Setting up an object's material properties and shininess

The subject of this section is the setting up of object's material. In this section instead of using the routine **glColor** in conjunction with colour tracking to create lighted objects, the more specific **glMaterial** will be used.

This routine will be used to specify the material's different components, diffuse, specular, emissive and ambient and how shiny objects are by setting the shininess. Because of the complex interaction between an object's material surface and incident light, specifying material properties so that an object has a desired, certain appearance is an art and is not something that can be learned from one moment to the other.

This routine, **glMaterial** accepts three arguments, the first being the face of the object that the material is going to be assigned, the second is the particular light component that needs to be set and the last one is a pointer to an array of values that will specify the appearance of the material (normally the array contains a red, a green, a blue and an alpha value). As mentioned before, the alpha value is used for blending and other 'special effects' and will not be used here. In the case of shininess the third parameter is not a pointer to an array but the actual value (0 to 128).

In this example the previously defined **Draw_Cube_Transl_Rot** will be slightly modified in order firstly to contain the appropriate material setting routines and secondly to draw a sphere instead of a cube (specular hilights are better shown on spheres, because of the larger amount of faces). Example 4.3 contains the code of this new function, called **Draw_Solid_Sphere**.

Example 4.3 Draw_Solid_Sphere function

This function accepts three arrays of type Glfloat. These arrays contain the values of the diffuse, specular and shininess components of the material. In this program, the emissive and ambient properties of the materials are not changed.

The body of the function should appear familiar. The only difference from the function **Draw_Cube_Transl_Rot**, being the addition of the three **glMaterial** calls. As seen in the example, both three calls set the front–face of the polygon to the specified values of the particular material property.

Back in the main program these three arrays are initialised to the values shown in example 4.4.

Example 4.4 The arrays containing the material properties values

GLfloat mat_diff[] = {1.0, 0.0, 0.0, 1.0}; GLfloat mat_spec[] = {1.0, 0.0, 0.0, 1.0};

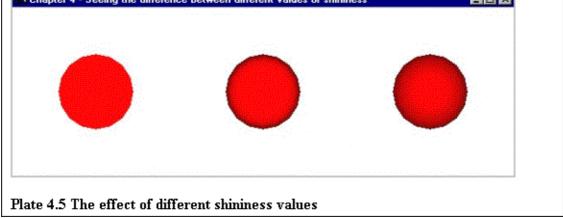
```
GLfloat mat_shin1[] = {0.0} ;
GLfloat mat_shin2[] = {5.0} ;
GLfloat mat_shin3[] = {50.0} ;
```

As seen in the example, the array *mat_diff*, that contains the values of the diffuse component of the material is set to red (1.0, 0.0, 0.0). The array *mat_spec* that contains the specular component values is also set to red. Three more array are specified called *mat_shin1*, *mat_shin2* and *mat_shin3*. These three arrays contain the shininess value of the three cubes that will be shortly drawn. Example 4.5 contains the code of the new display function. As it can be seen there, three viewports are defined and a red sphere of different shininess is rendered into each one of them. The results of this program can be seen in Plate 4.4.

Example 4.5 The display function that draws three red spheres with different shininess values

void display(void)

```
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ) ;
  glShadeModel(GL_SMOOTH) ;
  glViewport(0,0, win_size_H / 3 ,win_size_V) ;
  Draw_Solid_Sphere(mat_diff,mat_spec,mat_shin1) ;
  glViewport(win_size_H / 3,0, win_size_H / 3 ,win_size_V) ;
  Draw_Solid_Sphere(mat_diff,mat_spec,mat_shin2) ;
  glViewport(2*(win_size_H / 3),0, win_size_H / 3 ,win_size_V) ;
  Draw_Solid_Sphere(mat_diff,mat_spec,mat_shin3) ;
  glutSwapBuffers() ;
}
```



4.4 The Material – Lights program

The goal of this section is to create a program that a user will be able to see an object and how different materials and lights affect the appearance of the particular object. As this example is quite complicated, its discussion will be divided into several parts.

Firstly the appearance of the program will be considered. As the goal of this section is to show how different materials and lights affect an object, an object should appear on the screen. Also it should be clear by now that materials and lights are divided into deferent components. A material consists of four components (five if the shininess is included). These are the diffuse, the specular, the ambient and the emission. A light is also divided into components, and they are the diffuse, the specular and the ambient component.

All these components (except the shininess) are further composed from their red, green and blue elements. Some means of showing to the user the values of all these components and their elements should be found.

A solution was to create a function that would draw on the screen a graph, showing the red, green and blue values of a particular component. If now this function is used seven times, the four material components and the three light components could be visualised on the screen.

The problem is that orthographic projection should be used for drawing the graphs and perspective projection for drawing the objects. A solution to this problem would be to divide the window into several sub–windows, so that each sub–window could be assigned a different projection style.

It was then decided that eight sub-windows should be created. The main one would be used for drawing the objects and the other seven for drawing the seven material and light components.

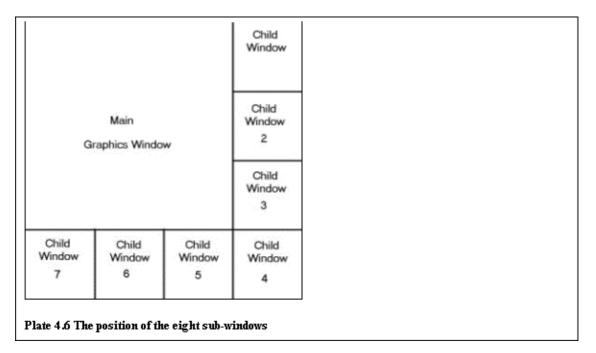


Plate 4.6 shows the positioning of the eight sub-windows.

The creation of the sub–windows can now start. *GLUT* provides a function named **glutCreateSubwindow** that can be used for this particular reason. This function accepts five arguments. The first one is the name of the parent window, the next two are the window initial x and y position and the last two are the window's width and height. Example 4.6 shows part of the main function that creates the main window and two of the sub–windows.

Example 4.6 Part of the main function that creates two sub–windows

```
parent_win = glutCreateWindow("Chapter 3 - Materials and Lights") ;
init_parent() ;
glutDisplayFunc(display_parent) ;
glutReshapeFunc(reshape_parent) ;
glutKeyboardFunc(keyboard) ;
glutSpecialFunc(special) ;
child_win = glutCreateSubWindow(parent_win, 3*(win_size_H/4), 0, win_size_H/4, win_size_V/4)
init_child_mat() ;
```

As it can be seen in this example, the main window is created using the familiar function **glutCreateWindow**. Any callback functions that are needed for the main window are registered and then the first sub–window, named child_win is created by calling the function **glutCreateSubWindow**. Two callback functions are registered to this sub–window (a display and a reshape one) and then another sub–window is created by using the same technique.

As seen in the example both sub–windows use the same reshape function. This function (shown in example 4.7) just creates an orthographic projection.

Example 4.7 The sub–windows reshape function

```
void reshape_child(int w, int h)
{
    glViewport(0, 0,(GLsizei)w,(GLsizei)h) ;
    glMatrixMode(GL_PROJECTION) ;
    glLoadIdentity() ;
    glOrtho(10.0,70.0,-10.0,110.0,-1.0,1.0) ;
    glMatrixMode(GL_MODELVIEW) ;
    glLoadIdentity() ;
}
```

Now that the sub–windows are created, it is time to create their contents. As mentioned before, an object will be drawn in the main sub–window to show the effect of assigning different values to the material and light components. In order for the user to see better these effects, eight different objects will be available to him. The function that will draw these objects can be seen in example 4.8.

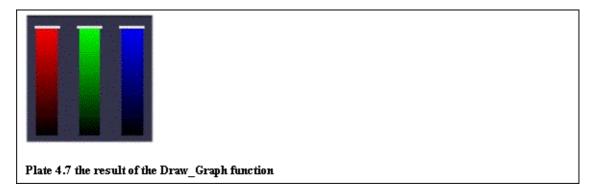
Example 4.8 Function that draws one of eight possible objects

```
void Draw_Object(int object)
{
    glPushMatrix() ;
    glTranslatef(tran[TRANS][0],tran[TRANS][1],tran[TRANS][2]) ;
    glRotatef(tran[ROTATE][3], tran[ROTATE][0], tran[ROTATE][1], tran[ROTATE][2]);
    glScalef(tran[SCALE][0],tran[SCALE][1],tran[SCALE][2]) ;
    switch (object)
    {
       case 1 :
          glutSolidSphere(1.0,32,32) ;
          break ;
       case 2 :
          glutSolidCube(1.0) ;
          break ;
       case 3 :
          glutSolidCone(1.0,1.0,32,32) ;
          break ;
       case 4 :
          glutSolidTorus(0.3,0.7,32,32) ;
          break ;
       case 5 :
          glutSolidOctahedron() ;
          break ;
```

Chapter 4- Lighting

```
case 6 :
    glutSolidTetrahedron() ;
    break ;
    case 7 :
      glutSolidIcosahedron() ;
      break ;
      case 8 :
      glutSolidTeapot(0.5) ;
      break ;
      default :
         break ;
    }
    glPopMatrix() ;
}
```

As seen in the example one of the following objects will be drawn to the screen depending on the value passed to the function: sphere, cube, cone, octahedron, tetrahedron, icosahedron, or teapot.



The function that will draw the seven graphs (one by one) must be constructed now. Plate 4.7 shows what this function is needed to draw.

As seen in the Plate, this function should draw a graph containing a red, a green, and a blue bar. Each bar stands for one of the three elements of the material and light components (red, green and blue). Each bar will have an index that will show the current value of the element. The code of this function named Draw_Graph can be found in Example 4.9.

Example 4. 9 The Draw_Graph function

```
void Draw_Graph(GLfloat Red_Height, GLfloat Green_Height, GLfloat Blue_Height)
{
    GLfloat Red[] = {1.0,0.0,0.0,1.0};
    GLfloat Green[] = {0.0,1.0,0.0,1.0};
    GLfloat Blue[] = {0.0,0.0,1.0,1.0};

    glPushMatrix();
    Draw_Bar(15,0,Red,Red_Height);
    Draw_Bar(35,0,Green,Green_Height);
    Draw_Bar(55,0,Blue,Blue_Height);
    glPopMatrix();

    glutPostRedisplay();
}
```

As seen in the example this function accepts three arguments. These arguments are the red, green and blue values of the component's elements that will be passed to the function **Draw_Bar** in order to position the index of the bars in the correct position. The function **Draw_Bar** accepts four arguments. The first two are

used to position the bar inside the window the third one to colour the bar and the last one to position the bars index in the correct place.

The bars are drawn by using smooth shading in order to draw the lower part black and the upper part the specified colour. Using this technique the index's position can approximate the colour of the component's particular element. Example 4.10 shows the code of the **Draw_Bar** function.

Example 4. 10 The Draw_Bar function

```
void Draw Bar(GLfloat x, GLfloat y, GLfloat color[], GLfloat height)
 {
    glPushMatrix() ;
    glBegin(GL_POLYGON) ;
    glColor3f(color[0] , color[1], color[2]) ;
                       ,y + 100.0) ;
    qlVertex2f(x
    glVertex2f(x + 10.0, y + 100.0);
    glColor3f(0.0, 0.0, 0.0) ;
    glVertex2f(x + 10.0,y);
    glVertex2f(x
                      ,y);
    glEnd() ;
    glPopMatrix() ;
    glPushMatrix() ;
    glTranslatef(0.0,height*100,0.0) ;
    glBegin(GL_POLYGON) ;
    glColor3f(1.0,1.0,1.0) ;
    qlVertex2f(x-1,y+2);
    glVertex2f(x+11,y+2) ;
    glVertex2f(x+11,y-1) ;
    glVertex2f(x-1,y-1) ;
    glEnd() ;
    glPopMatrix() ;
 }
```

As seen in the example this function is divided into two parts. The first part positions and creates a smoothly shaded rectangle while the second part uses the fourth argument of the function in order to position and draw the bar's index.

At this point nearly all the parts of the program used to demonstrate lighting effects are ready. A function, which still needs to be constructed, is the one that will be able to set the material and light properties. Actually two functions will be used for that purpose. The one named **Set_Material** will be responsible for setting up the object's material and the one called **Set_Light_ADS** will be used to set up the light components. Example 4.11 contains the **Set_Material** function and Example 4.12 contains the **Set_Light_ADS** function.

Example 4. 11 The Set_Material function

Example 4. 12 The Set_Light_ADS function

```
{
  glLightfv(light, GL_AMBIENT, ambient);
  glLightfv(light, GL_DIFFUSE, diffuse);
  glLightfv(light, GL_SPECULAR, specular);
}
```

The function **Set_Material** accepts six arguments. The first one, *pname* is used to specify the face to which the material is going to be applied. The other five arguments are arrays that contain the values that are going to be used in order to set the material up. This function uses the previously described routine **glMaterial** in order to set the various material components.

The function **Set_Light_ADS** (ADS stands for ambient, diffuse and specular) is similar to the function **Set_Material**. This time the function accepts four arguments. The first one is the light that is going to be set (i.e. LIGHT0) and the other three are arrays that contain the red, green and blue values of the diffuse, specular and ambient components of the light.

These arrays (containing the material and light components) are set inside the function keyboard. This function provides the needed keyboard interaction. The user can now manipulate the components and their elements by pressing several keys. Example 4.13 contains part of this function.

Example 4. 13 Part of the material-lights program keyboard function

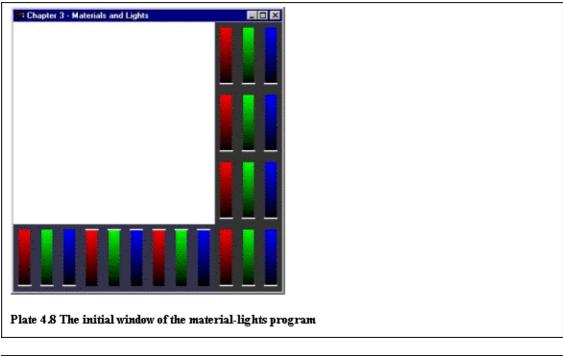
```
void keyboard (unsigned char key, int x, int y)
{
  static float steping = 0.05;
  switch(key)
  {
    case 'Q' :
      Increase(MATERIAL,AMBIENT,RED,steping);
      glutPostRedisplay();
      break;
      case 'q' :
      Decrease(MATERIAL,AMBIENT,RED,steping);
      glutPostRedisplay();
      break;
    }
}
```

As seen in the example the key 'Q' is used to increase the red element of the ambient component of the material by an amount equal to *steping*. 'q' is used to decrease the particular element by an amount equal to *steping*. The functions Increase and Decrease used here are two functions that increase or decrease the particular element of the particular component by an amount *steping*, making sure that the value of the element will not be greater than 1.0 or less than 0.0.

Plate 4.8 contains the program's window when initially run. The user can manipulate the various components by using the keys shown in Table 4.2. A function was also created in this program that is able to save the current material and light configuration in a file, for later reference.

The user can now 'play' with the material and light properties in order to understand how these can be combined to produce the needed colours, materials and effects.

This program can also be used to create a particular colour and then save it to the disk. For example, if a 'golden' colour is needed for a particular object in another program, a programmer can experiment with this program until the needed 'golden' colour is approximated and then he can save it and uses it in the other program.



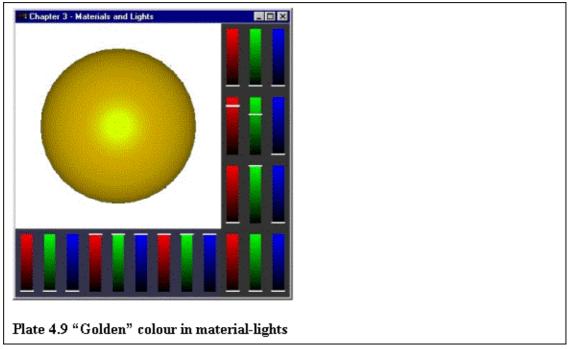


Plate 4.9 contains the window of the program after a user has created a 'golden' colour. The graphs on the right and lower part of the screen show the current values of the red, green and blue elements of the material and light components. The four graphs on the right part of the screen show the material components (from top to bottom) ambient. diffuse, specular and emission and the three graphs on the lower part of the screen (left to right) the light components ambient, diffuse and specular.

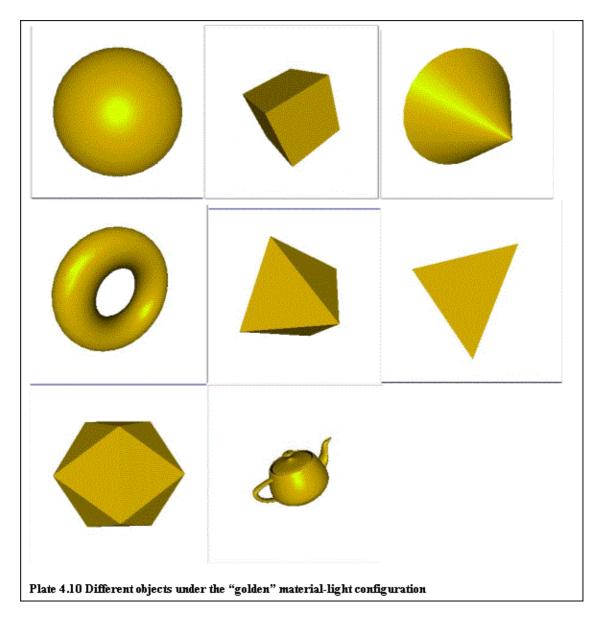


Plate 4.10 shows how deferent objects appear under the same material-light configuration.

Key	Material (M) or Light(L)	Component	Element	Action
2	M	Ambient	Red	Increase
7	M	Ambient	Red	Decrease
4	M	Ambient	Green	Increase
1	M	Ambient	Green	Decrease
Z	M	Ambient	Blue	Increase
5	M	Ambient	Blue	Decrease
W	M	Diffuse	Red	Increase
<i>N</i>	M	Diffuse	Red	Decrease
5	M	Diffuse	Green	Increase
3	M	Diffuse	Green	Decrease
x	M	Diffuse	Blue	Increase
(M	Diffuse	Blue	Decrease
3	M	Specular	Red	Increase
 9	M	Specular	Red	Decrease
 D	M	Specular	Green	Increase
1	M	Specular	Green	Decrease
2	M	Specular	Blue	Increase
2	M	Specular	Blue	Decrease
	M	Emission	Red	Increase
: :	M	Emission	Red	Decrease
7	M	Emission	Green	Increase
7	M	Emission	Green	Decrease
7	M	Emission	Blue	Increase
7	M	Emission	Blue	Decrease
6	M	Shininess	-	Increase
, 1	M	Shininess	-	Decrease
2	L	Ambient	Red	Increase
0	L	Ambient	Red	Decrease
	L	Ambient	Green	Increase
	L	Ambient	Green	Decrease
<	L	Ambient	Blue	Increase
	L	Ambient	Blue	Decrease
ſ	L	Diffuse	Red	Increase
-	L	Diffuse	Red	Decrease
-	L	Diffuse	Green	Increase
	L	Diffuse	Green	Decrease
>	L	Diffuse	Blue	Increase
	L	Diffuse	Blue	Decrease
	L	Specular	Red	Increase
	L	Specular	Red	Decrease
:	L	Specular	Green	Increase
	L	Specular	Green	Decrease
)	L	Specular	Blue	Increase
r r	L	Specular	Blue	Decrease

Table 4.2a

keys and their associated action (continued)							
1	Details affiliant						
2	Rotate object Animate object						
3	Cycle through objects						
5	Save material-light configuration						
6	Reset material						
7	Reset Light						
F1 to F8	Choose object						
ESC	Exit Program						

4.5 Adding lights to the basic model

This section is based on the program described in the second chapter, second section. This program is also data-driven, meaning that all its data are read from files.

For this reason two functions were used, named **Read_Data_From_File** and **Read_Material_From_File**. The first one (as described in Chapter 2) opens a file and reads the walking animation angles, the second one reads the body's materials. These functions will not be described here, as they contain only standard C calls in order to open a file and read some values.

The function **Read_Data_From_File** stores the data it reads into two variables of type *anim_angles* while the function **Read_Material_From_File** stores its data into variables of type *body_material*. The anim_angles structure was described back in Chapter 2, the custom type *body_material* is shown in Example 4.14.

Example 4. 14 The custom body_material

```
typedef struct
{
  float head[4][4] ;
  float head_j[4][4] ;
   float upbody[4][4] ;
  float lobody[4][4] ;
  float uparm_j[2][4][4] ;
   float uparm[2][4][4] ;
   float loarm_j[2][4][4] ;
   float loarm[2][4][4] ;
   float hand[2][4][4] ;
   float upleg_j[2][4][4] ;
   float upleg[2][4][4] ;
   float loleg_j[2][4][4] ;
   float loleg[2][4][4] ;
   float foot_j[2][4][4] ;
   float foot[2][4][4] ;
} body_materials ;
```

As seen in the example this structure is composed of floating point arrays. Body parts that appear twice in the body (like legs and arms) are arrays of dimension [2][4][4]. This array has these dimensions because a material has four components, diffuse, specular, ambient and emission ([2][4][4]), each component has four elements, red, green, blue and alpha ([2][4][4]) and the body has two of the particular parts, left and right ([2][4][4]). Parts that appear only once (like the head for example) are simply arrays of type [4][4].

Chapter 4- Lighting

Now that materials and angles are read and available to the program and the animation functions exist from the previously constructed program only the functions that draw the lit model remain to construct.

This is actually quite easy, as the only action needed to be taken is slightly modify the already ready modelling functions (constructed in the second section of the second chapter).

For this reason the previously constructed (previous section) function **Set_Material** is going to be used. Example 4.15 contains the **Draw_Head** function constructed back in the second chapter. This function does not contain the appropriate for lighting use routines, so a new function **Draw_Head** is going to be constructed containing a call to the function **Set_Material** in the appropriate point. Example 4.16 contains the new **Draw_Head** function.

Example 4. 15 The old Draw_Head function

```
void Draw_Head(int frame)
{
  glPushMatrix() ;
  glPushMatrix() ;
  glScalef(HEAD_WIDTH, HEAD_HEIGHT, TORSO) ;
  glColor3f(0.0,0.0,1.0) ;
   if (frame == WIRE)
     glutWireCube(1.0) ;
  else
      glutSolidCube(1.0)
  glPopMatrix() ;
  glTranslatef(0.0,-HEAD_HEIGHT * 0.66,0.0) ;
   glPushMatrix() ;
  glScalef(HEAD_JOINT_SIZE,HEAD_JOINT_SIZE,HEAD_JOINT_SIZE) ;
  glColor3f(0.0,1.0,0.0) ;
   if (frame == WIRE)
      glutWireSphere(1.0,8,8);
   else
      glutSolidSphere(1.0,8,8);
  glPopMatrix() ;
  glPopMatrix() ;
}
```

Example 4. 16 The new Draw_Head function

```
void Draw_Head(int frame)
{
  glPushMatrix() ;
  glPushMatrix() ;
  glScalef(HEAD_WIDTH, HEAD_HEIGHT, TORSO) ;
   if (frame == WIRE)
   {
      glColor3f(0.0,0.0,1.0) ;
      glutWireCube(1.0) ;
   else
   {
     Set_Material(GL_FRONT,material.head[0], material.head[1], material.head[2],
                 mat_shine,material.head[3]);
      glutSolidCube(1.0) ;
   }
  glPopMatrix() ;
  glTranslatef(0.0,-HEAD_HEIGHT * 0.66,0.0) ;
  glPushMatrix() ;
  glScalef(HEAD_JOINT_SIZE, HEAD_JOINT_SIZE, HEAD_JOINT_SIZE) ;
```

Chapter 4- Lighting

As seen in the example only a small, easy to identify part of the code needs to be changed (hi–lighted in yellow). After applying these changes to all the modelling functions the program is ready to be compiled and run.

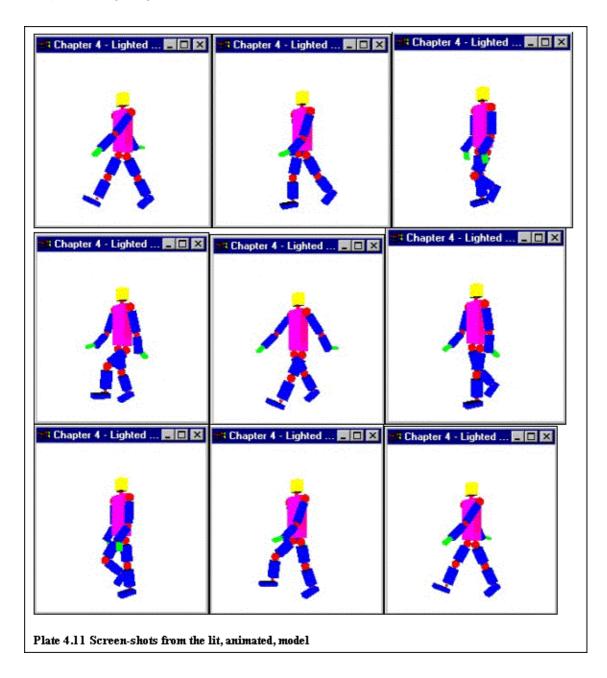
A new OpenGL feature is also used in this example in order to cut–down execution time. In the **init** function after the usual by now calls to **glEnable** with parameters GL_LIGHTING, GL_LIGHT0 and GL_DEPTH_TEST a new call can be found; that is **glEnable**(GL_CALL_FACE). When this value (GL_CALL_FACE) is passed to the routine **glEnable**, the OpenGL feature culling is enabled.

When culling is used all the back faced polygons are 'removed', meaning that no calculations are done concerning them and that they will not appear on the screen. That has the effect of cutting down to half the polygons a model is using, something that can dramatically increase the speed in cases of millions of polygons.

As the remaining of the program remains the same with the program discussed back in the second section of the second chapter, this program (and Chapter) can be considered finished.

After compiling and running this programs the results are the ones shown in Plate 4.11.

Chapter 4– Lighting



<u>Chapter 5 – Improving the model: "A more elaborate</u> <u>geometrical example</u>

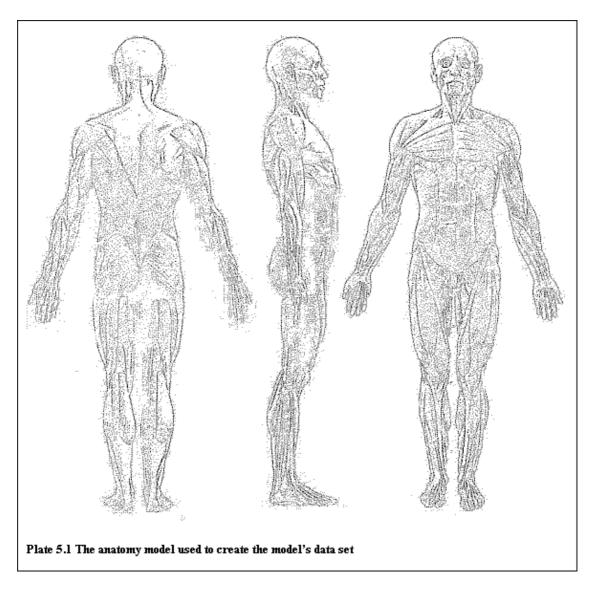
The topic of this chapter is the improvement of the basic model (constructed from spheres and cubes) discussed in chapter 2. Such a model may be good enough for demonstrating basic OpenGL concepts but it is not good enough for a commercial application that needs a model that approximates the human body, like a game, a virtual reality application, etc.

The data set for a human body was needed in order to construct a more elaborate example. Such data sets are available through the Internet; some of them free of charge, some not. Such data sets are normally constructed by scanning a three–dimensional object (in this case a human body). As three–dimensional scanners are quite expensive and there is always the possibility of not being able to find a ready–made model, it was decided that for this project a good modeling exercise would be to create the model's data set from scratch.

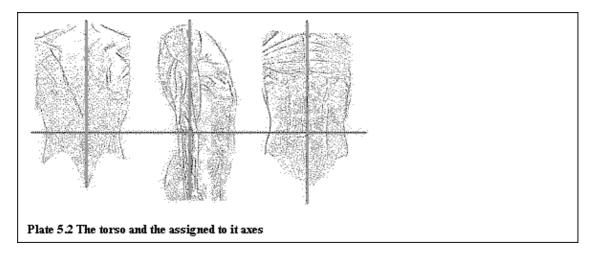
A technique had to be devised that would be able to create this three–dimensional model. In the first section of this chapter this technique is discussed, while the second part of the chapter contains a discussion of the program that reads the data set created in the first section and produces a three–dimensional model of a man.

After some research on the subject it was decided that a good technique was to try and analyze some conventional, two-dimensional photographs of a human body and try to retrieve the underlying three-dimensional model. A few anatomy books were consulted to find an appropriate model, and after some thought the model depicted in Plate 5.1 was chosen. Initially the model was not so clear, as in the process of photocopying and scanning, some noise was introduced to the image. This was corrected by applying the Paint Shop filter named find contour, resulting in the three images shown in Plate 5.1 (front, back and side view).

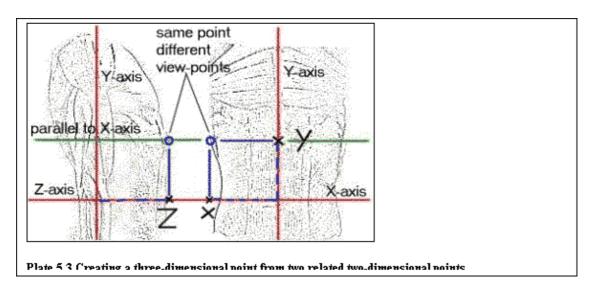
Chapter 5 - Improving the model: "A more elaborate geometrical example



Each one of these three images contains two-dimensional information about the model. By using two images in conjunction, three-dimensional information can be created. The first step was to design some reference axes. Plate 5.2 shows the torso and the appropriate axes that were assigned to it.



Chapter 5 – Improving the model: "A more elaborate geometrical example



At this point, it is quite easy to retrieve the data sets of the front and back part of torso (in two dimensions, x and y). By making lines parallel to the x-axis, the third dimension (depth, z) can be found for every single point. Plate 5.3 shows how to find the third dimension (z) of a random (x,y) point.

Applying this technique to an appropriate number of points can result in a very good quality, three–dimensional data set of the torso. If the technique is used on all body parts, the three–dimensional data set of a man model, will be constructed. This data set is going to be used in the next section of this chapter to create the OpenGL based model. Also, as seen in Plates 5.2 and 5.3 some parts of the body, like the torso have a degree of symmetry, so only half the points are needed (as the other half can be created by mirroring).

The goal of this section was not to create the full data set of a human model, but to show that the particular technique is working. As time was short, it was decided that it would be better to go on with the following parts of the projects than spending time finishing the data set of the model. In order to have enough data for the next section (the creation of the model) the points of the neck, torso and legs were retrieved.

5.2 Creating the complex model

The goal of this section is to use the points retrieved in the previous section to create a better–looking model of a man. As the body points are saved in a file some functions were created in order to load and store these points. These functions read the body points and store them in a structure named *body_points*. This structure is a set of multi–dimensional arrays, one for each body part. This structure is defined in the file general.h and it can be seen in Example 5.1.

Example 5.1 The structure body_points

```
typedef struct
{
   float neck[2][2][10][3] ;
   float torso[2][2][23][3] ;
   float upper_leg[2][2][23][3] ;
   float lower_leg[2][2][2][18][3] ;
} body_points ;
```

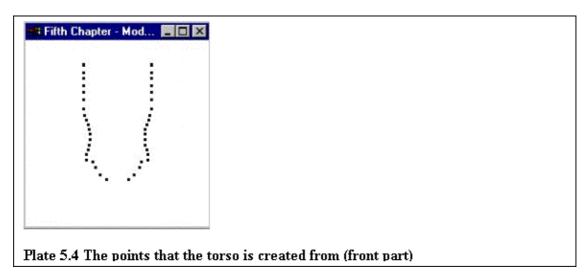
As mention in section 5.1 only the neck, torso and legs are going to be created in this section, so the custom type *body_points* contains an array for each one of these four parts. Starting from right to left, the first dimension is used to store the three–dimensional elements of the points (x, y, z), the second dimension to store the actual points (1^{st} , 2^{nd} , 3^{rd} and so on), the third dimension is used to distinguish between left and right side of a body part, the fourth to distinguish between front and back side of a body part, and when a fifth dimension is present is used to distinguish between left and right body parts (for example legs or arms).

Now that the structure were the points are going to be stored is explained, it is the time to discuss the functions that will read the points from the file.

These functions are implemented in the file named inout.c and the first one to be discussed is going to be the function **Return_Directory**. This function is going to be used in order to find the directory from which the program was executed so any needed files can be loaded from the same directory. This function accepts only one argument, a pointer to a character string. In every C/C++ program the first element of the *argv* array of the **main** function contains the directory from which the program was executed, including the name of the program (i.e. d:\programs\model\my_program.exe). The function **Return_Directory** takes this string and traverses it from right to left, until it finds a '\' character, then it returns the rest of the string, as the remaining part is the directory from which the program was executed. This information will be used later from other function to read any needed data.

The function **Read_Body_Points_From_File** does what its name suggests. This function accepts two arguments, the first one is the directory where the file should be and the second is a pointer to a *body_points* structure where the points will be stored. The directory of the file is found at run time by using the previously discussed function **Return_Directory** (the file should be at the same directory the program was executed).

As mentioned in the first section of this chapter not all the points were digitised only the key ones. Points that could be calculated by mirroring other points were not digitised. In the body of this function, after reading the digitised points from the file, the functions **Mirror_Data_Neck_Torso**, **Mirror_Data_Upper_Leg** and **Mirror_Data_Lower_Leg** are used to create the rest of the body points. As the neck and the torso have y-axis symmetry, the function **Mirror_Data_Neck_Torso** calculates the left-hand side points of both the front and back side of the torso by mirroring the right-hand side (digitised) points. The leg does not have y-axis symmetry between its left and right sides but it has y-axis symmetry as the whole part (the left leg is the mirror image of the right leg). So the functions **Mirror_Data_Upper_Leg** and **Mirror_Data_Lower_Leg** instead of mirroring points inside the leg, they are used to calculate the points of the left leg by inverting the points of the right leg.



Now that the points are read and available to the program, some function have to be created that will use these points in order to create the model. Plate 5.4 contains the results of drawing the points that construct the front part of the torso. These points can be connected in a variety of ways in order to create a solid model. As 3D accelerators usually accelerate models constructed from triangles, this approach will be followed.

When OpenGL lighting is used, the normal of the surfaces must be calculated. A normal is a vector that is perpendicular to the surface at a particular point and is used to calculate how light is reflected from the surface. Until now there was no reason to calculate the normals of the objects used, as these objects were created by using the available GLUT functions that contain also the needed normals.

A function was created that given three points of a three dimensional area that lies on the same plane in space (these points do not lie on a straight line), can calculate the unit normal to the surface.

This function calculates the perpendicular to the plane (the normal) by using the function

[v1-v2]x[v2-v3]

where the symbol 'x' means the cross product. v1, v2 and v3 are the three vectors that can be created when the three supplied points are joined.

Now that the function that calculates the normal to a surface is created, it is time to create the actual surfaces. As all the functions that create the body parts are similar, it was chosen to describe only one of them, the one that creates the torso. Example 5.4 contains this function.

Example 5.4 The Draw_Torso function

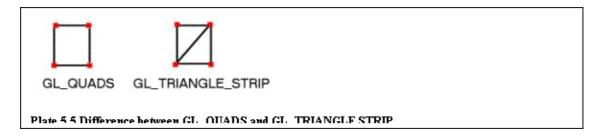
```
void Draw_Torso(int frame, float torso[2][2][23][3])
{
    glPushMatrix() ;
    if (frame == WIRE)
        glColor3f(1.0,0.0,0.0) ;
    else
        Set_Material(GL_FRONT,material.torso[0], material.torso[1], material.torso[2], material.s
        create_torso_front(torso[FRONT][LEFT], torso[FRONT][RIGHT]) ;
        create_torso_back(torso[BACK][LEFT], torso[BACK][RIGHT]) ;
        create_torso_sides(torso[FRONT][LEFT], torso[FRONT][RIGHT]) ;
        create_torso_sides(torso[FRONT][LEFT], torso[FRONT][RIGHT], torso[BACK][LEFT], torso[BACK][Front][LEFT], torso[FRONT][RIGHT], torso[BACK][LEFT], torso[BACK][Front][LEFT], torso[FRONT][RIGHT], torso[FRO
```

As seen in the example the structure of the **Draw_Torso** function is similar to the previous **Draw_Torso** function the difference being that instead of calling the GLUT functions to create the torso the custom made functions **create_torso_front**, **create_torso_back** and **create_torso_sides** are called.

The function accepts two arguments. The first one, *frame* can take the values WIRE or SOLID and is used in order to assign a colour or set a material (depending on if the model is wireframe or solid). The second argument contains the points that will be used to create the torso. Example 5.5 contains the function **create_torso_front** the other two functions will not be discussed here, as they are similar to this one.

Example 5.5 The create_torso_front function

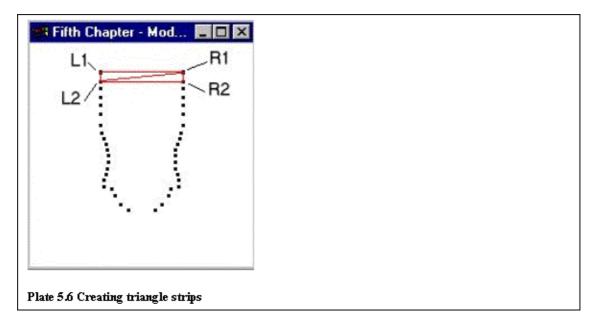
```
void create_torso_front(float left[23][3], float right[23][3])
{
    int counter ;
    float normal[3];
    for (counter = 0 ; counter <22 ; counter++ )
    {
        Calculate_Normal( left[counter],left[counter+1],right[counter+1],normal);
        glBegin(GL_TRIANGLE_STRIP) ;
        glNormal3fv(normal);
        glVertex3fv(left[counter]) ;
        glVertex3fv(left[counter+1]) ;
        glVertex3fv(right[counter]) ;
        glVertex3fv(right[counter+1]) ;
        glEnd() ;
    }
}</pre>
```



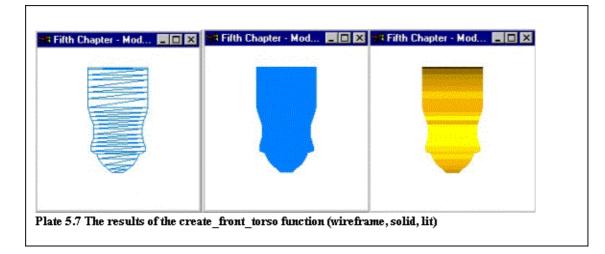
As seen in the example the body of this function is basically a **for** loop. Its time the for loop is executed, a normal is found by passing three appropriate values to the function **Calculate_Normal** and then the function **glBegin** is called with a GL_TRIANGLE_STRIP value. As explained previously the value passed to the function **glBegin** specifies what kind of object is going to be created. The value GL_TRIANGLE_STRIP is used to create triangle strips in the way shown in Plate 5.5.

The front part of the torso is constructed from 46 points. 23 of them are on the left-hand side and 23 on the right-hand side. The easiest way to create the front-torso surface using triangles is to create 22 triangle strips, each one of them constructed from 4 points. The way in which the points are selected is also important, as all triangles on the same surface should have the same orientation, in order not to have problems later when trying to use culling or a similar operation. Plate 5.6 shows the points and in which way they should be connected.

When four points are used to create a triangle strip, they should be specified in the order of: L1 à L2 à R1 à R2, as OpenGL will use vertices L1, L2 and R1 to create the first triangle and vertices R1, L2 and R2 to create the second triangle (in the exact order). In this way all triangles of a triangle strip are oriented in the same, anti–clockwise way.



Chapter 5 – Improving the model: "A more elaborate geometrical example



When the **for** loop finishes executing (22 times), the front part of the torso will be ready containing 22 triangle strips each one of them constructed from two triangles, giving a total of 44 triangles. The results of this function can be seen in Plate 5.7.

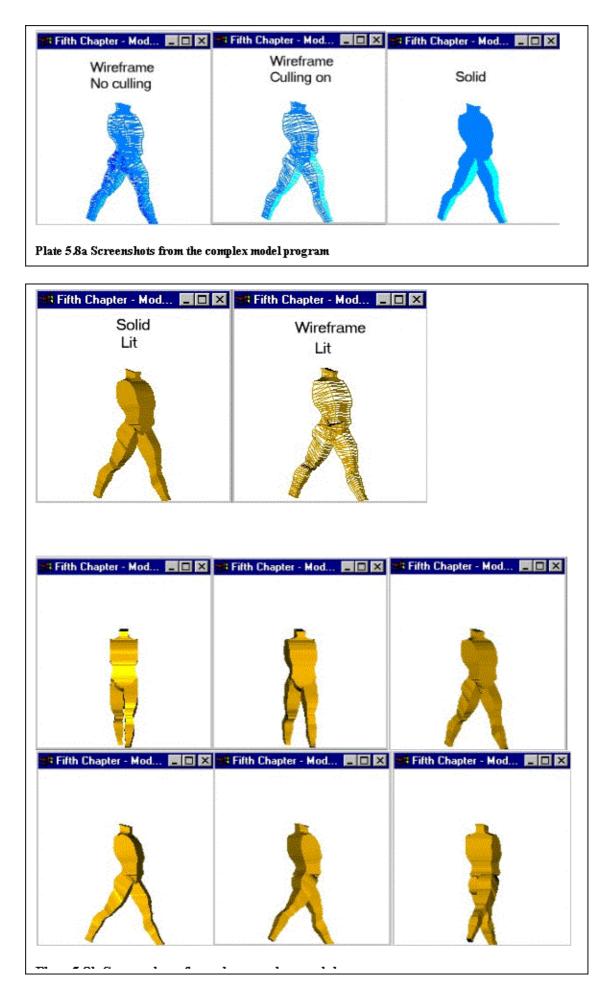
Using the same technique all the other modelling functions are created.

In order to animate the model a small change is needed in the **base_move** function, as the constants UP_LEG_HEIGHT, LO_LEG_HEIGHT and LEG_HEIGHT do not exist anymore. In order to solve this problem the two values UP_LEG_HEIGHT and LO_LEG_HEIGHT can be specified as externals and the value LEG_HEIGHT can be calculated (LEG_HEIGHT = UP_LEG_HEIGHT + LO_LEG_HEIGHT). The values of UP_LEG_HEIGHT and LO_LEG_HEIGHT are then calculated in the function **init**, in the main program, by finding the absolute value of the difference of the first and last points in the leg data set.

keys and their associated action								
a	Rotate object left							
s	Rotate object right							
х	Choose solid or wireframe							
с	Lights on/off							
v	Culling on/off							
1	Rotate in 3D							
2-8	Choose body part							
2	Animate model (walking)							

If this program is compiled and run the results will be the ones shown in Plate 5.8. Table 5.1 contains the keys used in the program and their associated operations.

Chapter 5 – Improving the model: "A more elaborate geometrical example



Until now, every geometric primitive has been drawn as either a solid color or smoothly shaded by interpolating the colors of its vertices. Texture mapping allows images to be glued on polygons and then follow the polygon's transformations.

With texture mapping, any image (scanned, drawn, etc.) can be applied onto a polygon, giving the polygon a completely different touch. Texture mapping ensures that acceptable things will happen to the image when the underlying polygon undergoes any transformations. For example if perspective projection is used in the scene any images used as textures will appear smaller as they get further from the viewpoint.

Texture mapping has many applications, some of them being wallpaper patterns, ground images in flight simulators, textures that make polygons look like natural substances such as marble, wood and so on.

Textures are simply rectangular arrays of data. The individual values in the texture array are often called texels. What makes texture mapping tricky is that rectangular textures can be mapped to non–rectangular regions, and this must be done in a reasonable way.

As texture mapping is such a large area, this chapter will not try to discuss the whole subject but explain the basics of texture mapping, how texture mapping is used in OpenGL and some of the basic filters that can be applied to textures.

As the application/example that was constructed for this Chapter is quite a lot more complicated than any of the previously discussed ones, its development will be divided into several sections.

Section one will discuss the functions needed in order to open and display a windows bitmap image (bmp) file. This image file was chosen because it does not use any compression when saving the image, so it is relatively easy to open and load the image.

As this program uses several windows, section two will discuss what has to be done in order to open and manage more than one window. Some functions of the Fast Light Tool Kit library are also explained in this section, as they are used in order to create the programs interface (buttons, pull–down menus etc.).

Section three continues and describes how a texture is created and then applied onto a polygon. As a simple example the texture is applied onto a cube.

The last section of this chapter explains the operations needed to put everything together, as well as describing the needed changes in the human model function to incorporate texture mapping.

In order to use texture mapping, some images must be available to the program. The easiest way to load and use an image as a texture is to save the image as a bmp file. This file format is very common among Microsoft Windows platforms and all image manipulation applications are able to save in this format. When the image is saved in the particular file format some function have to be created to load the image and make any needed manipulations to its data format in order to be of use with OpenGL.

Three functions were created for this reason (based on functions found in the book "OpenGL Super bible") named **LoadBitmapMy**, **ConvertRGB** and **SaveDIBitmap**.

The first function, **LoadBitmapMy** accepts two arguments and returns a pointer of type void. The first argument is a pointer to a character string that contains the directory and filename of the image. The second argument is a pointer to a BITMAPINFO structure. Every bitmap image contains a header of that type (BITMAPINFO), so when function **LoadBitmapMy** is called, the header of the image will be stored in a variable of type BITMAPINFO and the actual bitmap bits (each pixel of the image) will be stored in a pointer of type void.

This function is quite complicated, but there is no need to explain in depth what happens inside it. The main point is that an attempt is made to open the bitmap image file specified in the parameter *filename*; if this operation is successful, a check follows to discover whether the file is a bitmap file; if this check succeeds, memory is allocated for the bitmap header, the bitmap header is read and if everything is correct, memory is allocated for the actual image and the image is read. If everything went fine, the image is contained in the variable *bits* (of type void pointer) which is returned from the function.

The function **SaveDIBitmap** is similar to this one, as instead of opening a file and reading a bitmap into a void pointer variable, it receives an image in the form of a void pointer variable and saves it to the disk.

A problem with bitmap files is that the colour values are not saved in the order used by *OpenGL* (R–G–B), but in the order Blue, Green, Red. A function was needed that would swap the red and blue values of the image and that is the purpose of the **ConvertRGB** function.

At this point, after calling the two functions **ReadBitmapMy**, and **ConvertRGB** a bitmap image is available to the program (in the form of a void pointer variable). Now some appropriate OpenGL calls are needed in order to display this image in a window.

First of all, and in order to display the image correctly without any distortions, the projection used must be orthographic and the lower–left corner must be at (0,0) while the upper–right corner at (width –1, height –1) where width and height are the width and height of the image. This projection makes sure that the image will be displayed 'as–is' in a one–to–one way. Example 6.1 contains the **reshape** function that does that.

Example 6.1 The reshape function used to display an image

```
void reshape_main(int w,int h)
{
   glViewport(0,0,w,h) ;
   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   glOrtho(0, width-1,0, height-1, -1,1) ;
   glMatrixMode(GL_MODELVIEW) ;
   glLoadIdentity() ;
}
```

Now that the projection is set, is the time to do the actual drawing. The display function that is used to draw the image on the screen is shown in example 6.2.

Example 6.2 The display function used to display a bitmap image

As seen in the example after the colour buffer is cleared by calling the function **glClear**, a check is done to see whether the variable *BitmapBits* (used to store the image) contains any data.

If the variable is empty (NULL), nothing is done, otherwise the function **glPixelStorei** is called with the arguments (GL_UNPACK_ALIGNMENT, 4). This call specifies how data are going to be 'unpacked' from memory in order to draw it, in this case, by the function **glDrawPixels**. As the image is represented as a linked list of values (Red, Green, Blue, Alpha, etc.) OpenGL needs to know how to 'unpack' this data, meaning that the programmer should specify that for example every pixel is represented in the linked list by four values (R-G-B-A).

When this is done, a call to glRasterPos follows in order to set the current raster position.

As the orthographic projection used has its lower–left corner at point (0,0), the current raster position is set to the lower–left corner of the screen. This is needed because the function **glDrawPixels** starts drawing the lower–left part of the image at the current raster position and incrementally to the top–right.

glPixelStore parameters							
Parameter Name	Туре	Initial Value	Valid Range				
GL_UNPACK_SWAP_BYTES, GLPACK_SWAP_BUFFERS	GLboolean	FALSE	TRUE\FALSE				
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST	GLboolean	FALSE	TRUE\FALSE				
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	any nonnegative integer				
gl_unpack_skip_rows, GL_PACK_SKIP_ROWS	GLint	0	any nonnegative integer				
GL_UNPACK_SHIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	any nonnegative integer				
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT		4	1,2,4,8				

After the call to **glRasterPos** the current matrix is saved, the function glDrawPixels is called in order to draw the image and the matrix is restored.

The function **glDrawPixels** accepts five parameters. The first one is the width of the image to be drawn and the second is the height. The third parameter indicates the kind of pixel data elements to be used (Table 6.2) and the fourth one the type of each element (Table 6.3). The fifth parameter is a pointer to an array that contains the pixel data to be drawn.

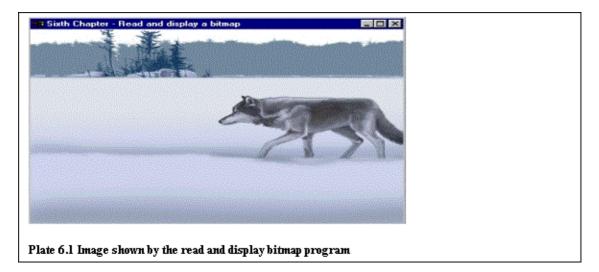
As seen in the example, the width and height of the image are passed to the function by using the bitmaps header. The format is set to R-G-B mode and the type to unsigned bytes. The array that contains the pixel data is the previously mentioned array *BitmapBits*.

The bitmap's header information (the width and the height) is also used in order to resize the window that the bitmap is drawn, in order to be the same size.

pidziafeypestfolog HBzsRRisksls							
GL_UNBORVENDBXTE	A single colour indensigned 8-bit integer						
GL_BCBE	A red colour componing to a blue						
GL_BCBMAP	Same as GL_RGB, follingle bibyinnulsign editionint gers using the						
GL_RED	A single red colour samepimenat as glBitMap						
GL_CINEEDNED_SHORT	A single green colo unsigned:térh it integer						
GL_SHORT	A single blue colousigment of finit integer						
GL HINEHCANED INT	A single alpha colossunignedmabit integer						
GL_INTMINANCE	A single luminance signed file thit integer						
GL_GUMANANCE_ALPHA	A luminance comparingter plice width flast interpretation component						
GL_STENCIL_INDEX	A single stencil index						
COMPONENT COMPONENT	A single depth component						

Tabla 6 1

Plate 6.1 contains some screenshots from images loaded using this program.



6.2 Opening several windows with OpenGL

This example will use several windows. One will be used for displaying the bitmap image from which the texture will be created (this window was the subject of the previous section). Another window will be used to show the texture (when this is created) and two more windows will be used to demonstrate texture mapping. One will be used for displaying a texture mapped cube and a second one to display the texture mapped model of a man.

The technique used to create the four windows is the same one employed in section 4, chapter 4. Example 6.3 shows part of the main function that creates the four windows and assigns to them any needed callback functions.

Example 6.3 Part of the main function that creates four windows

```
main_win = glutCreateWindow("Sixth Chapter - Texture Mapping") ;
glutHideWindow() ;
create_panel(argc,argv) ;
init() ;
glutDisplayFunc(display) ;
glutReshapeFunc(reshape_main) ;
glutKeyboardFunc(keyboard) ;
glutMouseFunc(mouse) ;
```

```
glutPassiveMotionFunc(passive_motion) ;
glutInitWindowSize(box_width,box_height) ;
glutInitWindowPosition(0,0) ;
texture_win = glutCreateWindow("Texture") ;
glutDisplayFunc(display2) ;
glutHideWindow() ;
init_cube_win() ;
glutInitWindowSize(200,200) ;
glutInitWindowPosition(0,100) ;
cube_win = glutCreateWindow("Distorted Cube Window") ;
glutDisplayFunc(display_cube) ;
glutReshapeFunc(reshape_cube) ;
glutSpecialFunc(special) ;
glutKeyboardFunc(keyboard) ;
glutHideWindow() ;
init_torso_win() ;
glutInitWindowSize(200,200) ;
glutInitWindowPosition(0,300) ;
torso_win = glutCreateWindow("Torso Window") ;
glutDisplayFunc(display_torso) ;
glutReshapeFunc(reshape_torso) ;
glutSpecialFunc(special) ;
glutKeyboardFunc(keyboard) ;
glutHideWindow() ;
```

As seen in the example, after the creation of every window (by calling the function **glutCreateWindow**) a call to the function **glutHideWindow** is issued. This GLUT function is responsible for hiding the current window, meaning that the window is created, but it is not visible to the user. This was done, because it was decided that no other windows other than the main interaction window should be visible to the user when the program is firstly run.

The main interaction window is created by calling the custom function **create_panel**. This function contains the needed Fast Light Tool Kit (FLTK) routine calls to create a FLTK window, containing some buttons and pull–down menus.

The problem with FLTK is that it is written in C++, so a C++ syntax must be used. After some thought it was decided that it would be overcomplicated to try and discuss the FLTK calls that are needed to create the user interface window, as the reader is used to the standard C conversions, so this window will be accepted 'as is'.

As seen in the previous example each window is assigned its own **display** and **reshape** functions. This was done because each window displays different graphics and thus needs different reshape conditions.

FLTK	_ 🗆 X				
Tongle Fu	Tongle Full Screen				
Tongle Bitr	Tongle Bitmap viewer				
Load Bitmap	Load Bitmap Save Bitmap				
kitom	Information				
Create Texture	Hide Texture				
Cube	Torso				
WRA WRA MAG_F MIN_F TEX	NP_T V				
J	kit				

A new GLUT routine also appears in this piece of code, named **glutPassiveMotion**. This routine is responsible for registering a passive motion callback function. What is meant by the term passive motion is that the mouse moves inside a window without any of its buttons pressed (active motion would therefore be the mouse does move when one or more of its buttons is pressed). This function is used in this program to animate a small selection box, when a texture is selected from a larger image.

6.3 Creating a texture

The topic of this section is the creation of a variable size texture. OpenGL textures can be of several different dimensions, depending to the implementation. Most OpenGL implementations support textures of dimensions up to 256 by 256 pixels (one-dimensional textures are possible but they are not discussed here). The size of two-dimensional textures must be a power of two (2x2, 4x4, 8x8 and so on).

In this program, the texture will be created by selecting a region of a (usually) larger image. A selection box will be rendered inside the image window (discussed in the first section of this chapter) and the user will be able to 'lock' the selection rectangle at some convenient to him point to create the texture. By locking it is meant that the selection rectangle will not follow the mouse movement from this point onwards.

The selection box is animated while the user moves the mouse inside the window using the callback function registered with **glutPassiveMotion**. When a convenient place is found the user can press the right mouse button to 'lock' the selection rectangle and then create the texture.

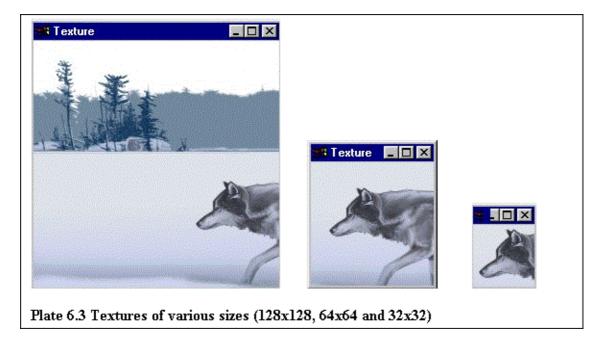
The texture is created by calling the function **show_texture_cb**. This function is shown in example 6.4.

Example 6.4 The function show_texture_cb

```
void showTexture_cb(Fl_Widget *, void *)
{
    NewBitmapBits = array ;
    glReadPixels(mouse_coX,height - mouse_coY, box_width, box_height, GL_RGB,GL_UNSIGNED_BYTE,
    glutSetWindow(texture_win) ;
    glutReshapeWindow(box_width,box_height) ;
    glutShowWindow() ;
    glutPostRedisplay() ;
    glutSetWindow(main_win) ;
}
```

Actually the texture is not created at this point but later on; what happens in this function is that the function **glReadPixels** is used to read the pixels which are under the selection area into the variable *array*. The routine **glReadPixels** has exactly the opposite effect of the previously explained routine **glDrawPixels**.

When the pixels inside the selection box are stored in the variable *array*, the routine **glutSetWindow** is used to set the texture window as the current, then the texture window is resized to the dimensions of the texture. Finally the window is shown.



The size of the texture created depends on the size of the selection box. The size of the selection box (and the texture's) can be set by calling the function **texturesize_cb**. The body of this function is actually a switch **statement** and each time the function is called a flag is incremented, thus cycling among the predefined texture sizes (one of the **switch** cases). Plate 6.3 contains various sizes textures, created by this method.

The actual texture, as mentioned before, is not created in the function **create_texture_cb**, but in the body of the **display** function that creates the texture mapped cube.

In the body of this function, after the usual function calls (**glClear** etc.), the routine **glEnable** is called with the value GL_TEXTURE_2D passed to it. This call enables OpenGL's texture mapping ability.

possible combination of values for the routine glTexParameter							
GL TEXTURE WRAP S	GL CLAMP, GL REPEAT						
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_REPEAT						
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR						
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR,						
GL_TEXTURE_BORDER_COLOR	any four values in the range [0.0, 1.0]						
GL_TEXTURE_PRIORITY	[0.0, 1.0] for the current texture object						

The next routine appearing for the first time in this function is **glTexParameter**. This routine is responsible for setting various parameters that control how a texture is treated and it accepts three arguments; the first argument can be either GL_TEXTURE_2D, or GL_TEXTURE_1D to indicate a two- or one-dimensional texture. The possible combinations of values for the second and third parameter are shown in Table 6.4. Visual examples of the effect of this function will be found near the end of this section.

Back in the body of the **display_cube** function, just after the calls to **glTexParameter**, a call to **glTexEnv** is issued. The purpose of this function is to specify how the texture colours are going to be calculated. The colour of a texture can be the colour of its own texels, or a combination of its own colour and the surface on which it is applied. Visual examples of the effects of this routine will appear at the end of this section.

Following, the routine **glTexImage2D** is called. This is the most important routine in the **display_cube** function, as it is the one that defines the actual two–dimensional texture. This function accepts quite a few arguments, therefore an example is given here to explain what each one is used for.

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, box_width, box_height, 0, GL_RGB, GL_UNSIGNED_BYTE, array);

The first parameter can be either GL_TEXTURE_2D or GL_PROXY_TEXTURE_2D. In this report the constant GL_PROXY_TEXTURE_2D is not discussed, so GL_TEXTURE_2D is used. The second parameter is used when supplying multiple resolutions of the texture. Multiple texture resolutions are used for mip-maping, something that is not covered in this report, so this parameter is set to 0 (only one resolution).

The next parameter indicates which of the R, G, B, and Alpha components or luminance or intensity values are selected for use in describing the texels of the image. This can be one for thirty eight symbolic constants. The one used here, GL_RGB, specifies that the Red, Green and Blue components are used to describe the texel.

The next two parameters specify the width and the height of the texture, and as described before they are in this case the width and the height of the selection box. The next parameter is the border of the texture and can be either 0 (no border) or 1. Both the width and the height of the texture must have the form $2^m + 2b$, where m is a nonnegative integer and b is the width of the border. In this example no borders are used (0 is passed to the function).

The following two parameters describe the format and type of the texture image data, and they have the same meaning as in the case of **glDrawPixels** (Tables 6.2 and 6.3), with the exception of GL_STENCIL_INDEX and GL_DEPTH_COMPONENT (for the format parameter).

Finally the last parameter contains the texture-image data. These data describe the texture image itself as

well as the border.

At this point the texture environment, the appearance and the texture itself are constructed, set and ready to use. It will now be discussed what texture co–ordinates are and how they should be specified.

In OpenGL, textures are treated as normal objects, so it is typical to be able to set their co-ordinates. When texture mapping is used, both object and texture co-ordinates must be provided for each vertex. After transformation, the object co-ordinates determine where on screen that particular vertex is rendered. The texture co-ordinates determine which texel in the texture map is assigned to that vertex. Texture co-ordinates are interpolated between vertices in the same way colour values were.

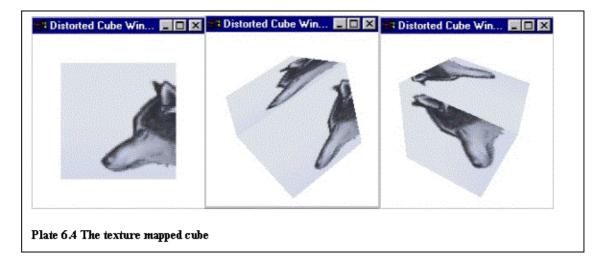
Depending on the texture co-ordinates applied, the texture can be mapped one-to-one, inverted, stretched, shrunk, etc. Certain visual examples will appear later on to help visualise the concept.

For the moment, every vertex of the cube is assigned either a 0 or 1 texture co-ordinate (inside the **display_cube** function). This results in the (square) texture, mapped one-to-one to each (square) face of the cube.

At this point the **display** function **display_cube** that will be used to draw a texture mapped cube is ready. Plate 6.4 contains some screenshots of the texture mapped cube (the default values of environment and texture parameters are used).

Plate 6.5 demonstrates the effects of the function **glTexEnv**, as it contains screenshots with different environment settings and Plate 6.6 shows the cube under the effect of different texture mapping filters (**glTexParameter**).

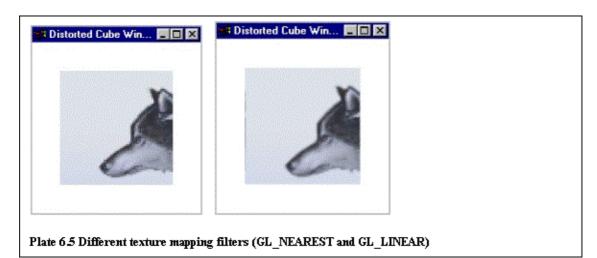
The last plate in this section, Plate 6.7 shows what happens when different texture co-ordinates are used (in conjunction with the **glTexParameter** parameters GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T). For this reason the function **display_cube** was slightly modified, and instead of setting the texture co-ordinates inside the function, global variables are used as the texture co-ordinates, which can be changed with the keyboard's directional keys (arrows).



In this project global variables are used in some occasions, but not as a result of 'bad programming practice' but because functions like this one (**display**) are of type **void**, meaning that they can not receive any parameters.



Plate 6.5 Different texture environment values (GL_MODULATE and GL_BLEND)



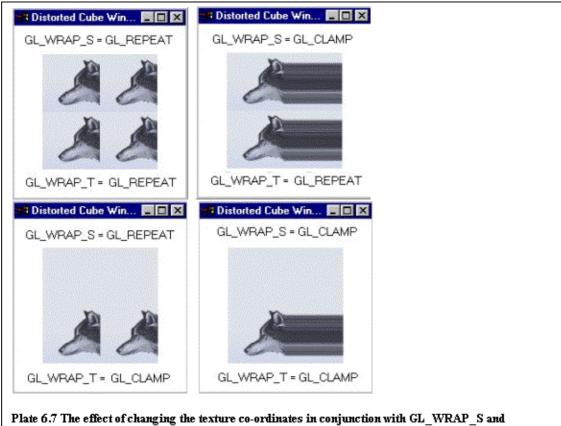


Plate 6.7 The effect of changing the texture co-ordinates in conjunction with GL_WRAP_S GL_WRAP_T

6.4 A texture mapped man

The goal of this section is to apply the texture mapping techniques discussed in the previous section onto the model created in Chapter 5. For that reason the modelling functions constructed in Chapter 5 (**Draw_Leg**, **Draw_Arm**, etc.) will have to be modified to include texture co–ordinates statements.

The same texture will be applied to the whole body, so any texture construction and setting will be done in the main program. The modelling functions will only have to include the appropriate texture co-ordinates statements.

As mentioned in the previous section, every object must have its texture co-ordinates assigned appropriately, otherwise the texture will be so distorted that will be unrecognisable.

The same texture will be applied to all the parts of the body, once on each side (i.e. the texture will appear four times on the torso at its front, back, left and right side). As the body parts are not square and they are constructed from many vertices, a way must be found to calculate the texture co–ordinates for every vertex.

Every 'side' (front, back, left or right) is constructed from two arrays, the 'left' and the 'right' (refer to Chapter 5). The 'horizontal' texture co-ordinates are quite easy to assign, as no calculation is involved. If every vertex in the 'left' array is assigned a x (horizontal) texture co-ordinate of 0 and every vertex in the 'right' array a x co-ordinate of 1, a nice, 'tight clothes effect', can be achieved (as the texture will be shrunk, the impression of clothes tight to the body will be given).

The problem appears when trying to assign the 'vertical' (y values) co–ordinates of the vertices. It is clear that the lower point of every part will be assigned the value 0 and the higher point the value 1, but how can the in–between values be calculated?

Every vertex in the array has three values, a x, an y and a z value. The appropriate y texture co-ordinate can be calculated by the following technique:

- the higher vertex of the body part is assigned a texture value of 1 (y value)
- the distance between this point and the next point is found
- this distance is divided by the total body part length

 $\cdot\,$ the next (lower) vertex of the body part is assigned a value of 1 minus the calculated value ((point – next_point) / length)

 \cdot the technique, is repeated until all vertices have texture co-ordinates assigned to them, the last vertex of the body part will have a value of 0.

A demonstration of this technique, can be found in example 6.5, where the function that draws the front part of the torso is shown.

Example 6.5 The function create_torso_front (texture mapped)

```
void create_torso_front(float left[23][3], float right[23][3])
{
    int counter ;
    float normal[3] ;
    float texLenght = abs(left[0][1])+abs(left[22][1]),
        texCooUp = 1,
        texCooDown = 0 ;
    for (counter = 0 ; counter <22 ; counter++ )</pre>
```

```
{
    Calculate_Normal( left[counter], left[counter+1], right[counter+1],normal)
    texCooDown += abs((abs(left[counter+1][1]) - abs(left[counter][1])))/texLenght ;

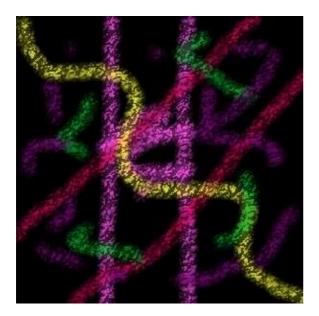
    glBegin(GL_TRIANGLE_STRIP) ;
    glTexCoord2f(0.0,texCooUp) ; glVertex3fv(left[counter]) ;
    glTexCoord2f(0.0,1-texCooDown); glVertex3fv(left[counter+1]);
    glTexCoord2f(1.0,texCooUp); glVertex3fv(right[counter]) ;
    glTexCoord2f(1.0,1-texCooDown); glVertex3fv(right[counter+1]);
    glEnd() ;
    texCooUp = 1-texCooDown ;
}
```

When all modelling functions are changed the program is ready. Plate 6.8 contains the finished program. The user can manipulate the various buttons and pull-down menus in order to load a picture, create a texture, save a texture and so on. Table 6.5 contains a description of the components of the user interface and their associated actions. Plate 6.10 contains some screen shots from the final version, while Plate 6.9 shows the texture that was used to create the results shown in Plate 6.10.

	The user interface of the texture mapping program				
Toggie Full Screen	Switches between fullscreen and normal view (Image viewer)				
Foggle Bitmap Viewer	Switches the bitmap viewer on and off (visible – invisible)				
Load Bitmap	Shows a file browser and loads a bitmap image				
Save Bitmap	Saves a texture as a bitmap file				
Information	Shows some copyright information				
Create Texture	Shows the selected image area in a separate window				
Hide Texture	Hides the texture window				
Cube	Displays the rotating, texture mapped cube (the spacebar toggles the animation on and off)				
Torso	Displays the texture mapped torso (Previously described keyboard interaction applies also in this example)				
WRAP_S	Selects a horizontal wrapping type (GL_CLAMP or GL_REPEAT)				
WRAP_T	Selects a horizontal wrapping type (GL_CLAMP or GL_REPEAT)				
MAG_FILTER	Selects a magnification filter				
MIN_FILTER	Selects a diminution filter				
TEX_ENV	Selects a texturing function				
Change Size	Changes the selection box size (as well as the texture's size)				
Exit	Exits the program				

FLTK							
Tongle Full Screen							
Tongle Bitmap viewer							
Load Bitmap Save Bitmap							
Inform	ation						
Create Texture	Hide Texture						
Cube	Torso						
WRA	P_S T						
WRA	P_T						
MAG_F							
MIN_F							
TEX	ENV V						
Change Size							
Ex	(it in the second						

Plate 6.8





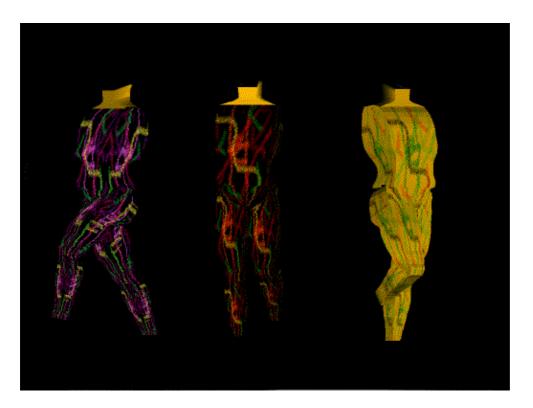


Plate 6.10

Chapter 7 – Conclusions – Future possibilities

This project constituted an introduction to OpenGL and three–dimensional graphics. It was an effort for the writer to learn and at the same time attempt to design a tutorial on this subject, so that future readers will be able to follow his work and possibly expand it.

This paper began with the discussion of the basics of both three–dimensional graphics and the OpenGL structure. The introduction contained the theoretical framework needed for the project, including the reasons for the specific structure followed. The topic of the second chapter was simple window construction, as OpenGL needs a graphical (windowing) operating system, and the introduction of modelling and projection transformations. In the third chapter a first attempt was made to create a simple model of a man and the appropriate animation cycle, which resulted in a model constructed from basic geometrical shapes (spheres and cubes).

The fourth chapter introduced OpenGL's lighting model and continued with the discussion of materials and their properties. A program was constructed where a user can experiment with the light and material properties in order to familiarise with the concept. A more elaborate geometrical example was presented in the fifth chapter, as its discussion topic was the improvement of the basic, until now, model. The sixth chapter introduced texture mapping, a technique that enables the use of images as parts of objects, making OpenGL programs more attractive. The topic of texture mapping has not been thoroughly exhausted, as the subject is quite complicated and the applications of texture mapping are inexhaustible.

During the specific time limits that were set for this project, all its primary objectives were accomplished. Given more time, further elaboration and 'special effects' could have been achieved. The list of those is practically unlimited but some ideas include shadows, fog, blending, collision detection, 'selection and feedback', and possibly the use of the DirectX component Direct Sound for sound effects. Actually, research was made on the previous two topics but it was not published in this paper as completion was not achieved.

This project was a good opportunity for the writer to be introduced in long scale, real life problems in opposition to the academic, small scale practical coursework. It would be an accomplishment if the present paper manages to assist people who wish to make a start with three–dimensional graphics and OpenGL.

Appendix I – Using Borland C++ 5.02

The main environment used in the development of this project was Borland C++ 5.02. In order to open windows using the OpenGL tool kit (GLUT) the C project must be specified as WIN32. The problem is that a WIN32 project is not as simple as a DOS C program, as the main function is not the one used anymore. The best way to open windows using GLUT is to build a WIN32 console project. This type of project still uses the main function as its basic function but has all the advantages of a WIN32 application. The following steps are needed in order to build such a project in Borland C++ 5.02.

File	Edit	Search	⊻iew	Project	Script	Tool	Debug	<u>Options</u>	<u>₩</u> indow	Help
Ne	ew							<u>I</u> ext Edi	it	Ī
<u>0</u> p	oen							Project.		
<u>C</u> k	ose							AppExp	ert	
Sa	ive				C	trl+K Ct	rl+S	<u>R</u> esourc	e Project	
Sa	ive <u>a</u> s.	-					Т			
Se	ive all									

Figure I. 1

Run Borland C++ 5.02. Wait until the environment is loaded and select : <u>New</u> \rightarrow <u>Project</u> (figure I.1).

When this is done a window with several options will appear (figure I.2). In the sub-window named <u>Project</u> Path and Name, type in the path of the project (or press browse and select a path). In <u>Target Name</u> type in the project name. At sub-window <u>Target Type</u> select <u>Application[.exe]</u>. At the sub-window <u>Platform</u> select Win32 and at <u>Target Model</u> select Console. Do not change any other options and press OK.

The new project is ready (figure I.3) !

w Target		?
Project Path and Name:	- 🗸 ок	
e:\opengl related staff\pui\my_new_p		
Target Name:	🗶 Cancel	
my_new_project		to- Browse
Target Type: Application [.exe] Dynamic Library [.dll] EasyWin [.exe] Static Library (for .exe) [.lib] Static Library (for .dll) [.lib] Import Library [.lib] Platform: Win32 Target Model: Console	Frameworks: Class Library □ MFC OCE 0 32 OWL 0 4× Controls: BWCC □ VBX CTL3D Libraries: OLE	Advanced ? Help
	Djagnostic Multithread	

Figure I. 2

🕼 Barland C++ - my_new_project 📃									- E >	
Eile	Edit	Search	⊻iew	Project	Script	Tool	Debug	Options	Window	Help
						in the				📴 Project : e:\bp\temp\my 🗖 🗖 🔀
										• E 📑 🖉 😽 🖌 ny_nev_project.exe
										 ay_new_project.cpp [
										 ny_nev_project.def [
										• _ ny_nev_project.rc [.

Figure I. 3

Delete any files the environment has created (such as my_new_project.cpp) by pressing the right mouse button on the file name and choosing <u>D</u>elete node.

Now insert the needed files for the project by pressing the right mouse button on the project name and selecting <u>Add Node</u> (figure I.4).

Project : e:\bp\temp\my_new_project.ide						
• — 🗆 🏶 🖌 my_new_project.ex	e [exe] View					
	Add node					
	Delete node Make node Preview make Build node Link Special	•				
	<u>I</u> argetExpert Edit <u>n</u> ode attributes Edit local <u>o</u> ptions View options <u>h</u> ierarchy P <u>r</u> operties					

Figure I. 4

A browser window will appear, choose all the needed files (such as main.c etceteras). Insert also the files opengl.lib, glu.lib and glut.lib. These three libraries are the ones needed in order to use OpenGL. The environment is now going to look something like figure I.5.

e	Edit	Search	View	Project	Script	Icol	Debug	<u>Options</u>	Window	Help	
											Project: e:\bp\temp\my Project: e:\bp\temp\my

Figure I. 5

The project is now ready. The program can be executed by pressing <u>Debug -> Run</u>.

If you do not have the borland libraries, you can create them using the following procedure:

Find the opengl and glu dynamic link libraries (.dlls). These come normally with your graphics card drivers. When you have located them copy them in a temporary directory, and use borland's command line program 'implib'.

This program takes as input a dynamic link file (dll) and produces the corresponding library file (lib).

Appendix II – Using The FLTK Library

FLTK (Fast Light Tool Kit) is a GUI (graphical user interface) for UNIX (X–Windows) and Windows (95/98/NT) and is fully compatible with OpenGL. Because of its compatibility with both windows systems it was used for the creation of buttons and other widgets for some of the programs created in this project.

The following example of how to use FLTK to build a simple FLTK window with a box (widget) inside it saying hello world is taken from the FLTK help file.

More examples on using FLTK (taken from the created OpenGL programs) and the specification of the FLTK tool kit will appear in the website (<u>www.dev-gallery.com</u>).

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>
int main(int argc, char **argv)
{
    Fl_Window *window = new Fl_Window(300,180);
    Fl_Box *box = new Fl_Box(FL_UP_BOX,20,40,260,100,"Hello, World!");
    box->labelsize(36);
    box->labelfont(FL_BOLD+FL_ITALIC);
    box->labeltype(FL_SHADOW_LABEL);
    window->end();
    window->show(argc, argv);
    return Fl::run();
}
```

All programs must include the file $\langle FL/Fl.H \rangle$. In addition the program must include a header file for each FLTK class it uses, here $\langle FL/Fl_Window.H \rangle$ and $\langle FL/Fl_Box.H \rangle$.

The program then creates a window and then creates the widgets inside the window. Here a single Fl_Box is created. The arguments to the constructor are a value for the box() property (most constructors do not have this), values for x(), y(), w(), h() to define the position and size of the box, and a value for label() to define the text printed in the box.

All the widgets have several attributes and there is a method for setting and getting the current value of each of them. box->labelsize(36) sets the labelsize() to 36. You could get the value with box->labelsize().

Often you have to set many properties, so you will be relieved to know that almost all of these methods are trivial inline functions.

labelfont() is set to a symbolic value which is compiled into a constant integer, 3 in this case. All properties that cannot be described by a single small number use a 1–byte index into a table. This makes the widget smaller, allows the actual definition of the property to be deferred until first use, and you can redefine existing entries to make global style changes.

labeltype(FL_SHADOW_LABEL) also stores a 1-byte symbolic value, in this case indicating a procedure to draw drop shadows under the letters should be called to draw the label.

The constructor for widgets adds them as children of the "current group" (usually a window). window->end() stops adding them to this window. For more control over the construction of objects, you can end() the window immediately, and then add the objects with window->add(box). You can also do window->begin() to switch what window new objects are added to.

window->show() finally puts the window on the screen. It is not until this point that the X server is opened. FLTK provides some optional and rather simple command-line parsing if you call show(argv, argc). If you

Appendix II – Using The FLTK Library

don't want this, just call show() with no arguments, and the unused argument code is not linked into your program, making it smaller!

Fl::run() makes FLTK enter a loop to update the screen and respond to events. By default when the user closes the last window FLTK exits by calling exit(0). run() does not actually return, it is declared to return an int so you can end your main() function with "return Fl::run()".

Appendix III – Using Paint Shop Pro 5.0

The task of chapter four was to improve the simple (in chapter one) model of a human into something better than just rectangles and spheres. In order to do something like that data were needed in some form of a human body.

In the Internet there many resources of freely available data sets of human models. Another less straight forward way was chosen in this project. If the ready made data (from the Internet) were used the developer of this project would not know how to retrieve such data.

Nowadays three dimensional scanners are available but their price is so high that they are not yet massively available. In such a case that somebody does not posses a 3D scanner but has to model an object, and the objects data set is not freely available in the Internet, then the question is what happens?

In this project the pessimistic (but realistic) approach was chosen that the developer does not have any access to a 3D scanner and that the data set of the object is not available. In this case other means of retrieving the data of a three dimensional object have to be found.

One such technique is the one described in the following lines. For the purpose of this task, the painting program Paint Shop Pro 5.0 was used.

Firstly, three photographs of a human male body were scanned from a book on anatomy [5-10], a front, a back and a side view. These three photos were then put into Paint Shop Pro (figure III.2). Four more layers were created on top of the basic one (the background) these four layers held the following data :

- \cdot 0 (background) the picture
- 1 black background (invisible in the beggining)
- · 2 vertical rulers
- · 3 horisontal rulers
- · 4 points

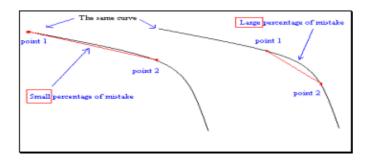


Figure III. 1

By using the mouse points were drawn (on the points layer) wherever a curve changed direction (in order that two consecutive points could form a line without a big percentage of loss of information) figure III.1.

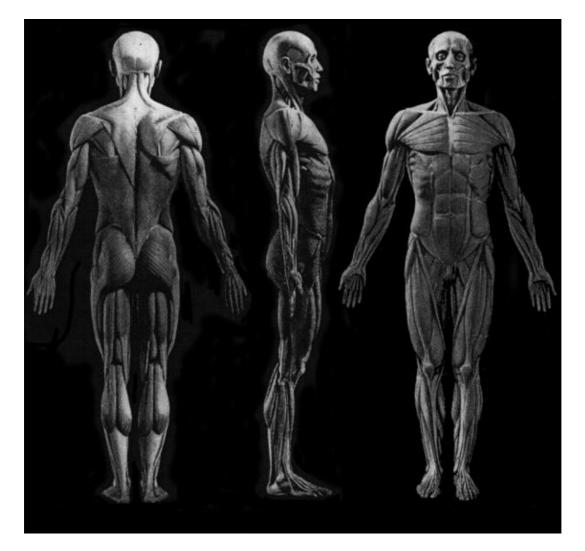


Figure III. 2

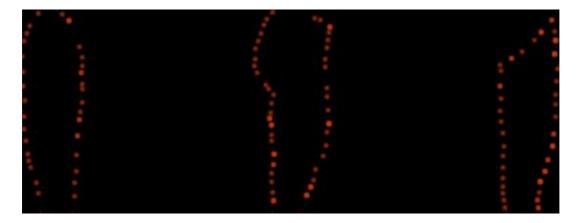


Figure III. 3

By following the technique demonstrated at figure III.1 the points of the body were retrieved from the three two dimensional images (figure III.4). By disabling all layers except the points and the black background layer the points can be observed quite better (without any other confusing information) (figure III.3).

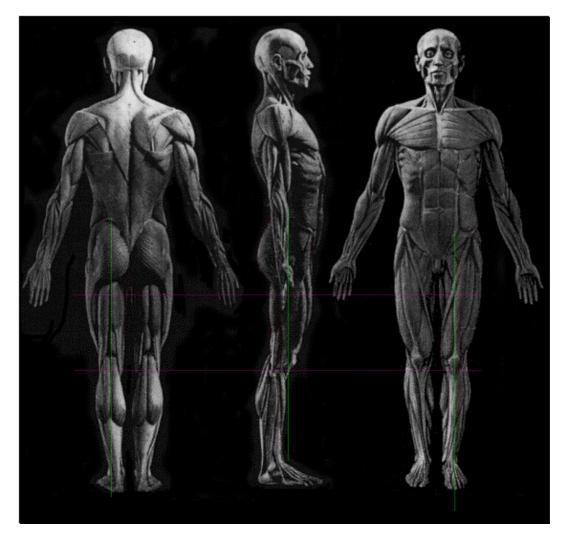


Figure III. 4

By working on several pictures like figure 3 the relations between height width and depth can be retrieved (further information in the final report).

The task is completed! A three dimensional image has been created by the manipulation of several two dimensional ones.

Note : The photocopies of human bodies used in this project were found in Farnham, Surrey (Library of the College of Arts).

Appendix IV – Bibliography

- [1] **Moving Pictures** (In Greek) Tony White
- [2] **The Male And Female Figure In Motion** Eadweard Muybridge Dover Publications, Inc New York
- [3] OpenGL Programming Guide (Second Edition) OpenGL Architecture Review Board Mason Woo, Jackie Neider, Tom Davis ISBN 0-201-46138-2
- [4] **OpenGL Superbible** Richard S. Wright Jr, Michael Sweet ISBN 1–57169–073–5
- [5] The Human Machine (The anatomical Strucure And Mechanism Of The Human Body)
 George B.Buidgman
 Dover Publications, Inc
 New York
- [6] Anatomical Man, Bones And Muscles For The Student Silvio Zaniboni London Alec Tiranti 1963
- [7] **Illustrator's Figgure Reference Manual** Bloomsbury ISBN 0-74750-008-8
- [8] Anatomical Diagrams For The Use Of Art Students James M.Dunlop A.R.C.A.
 G.Bell & Sons Ltd, York House Portugal Street
 W.C.2 MDCCCCLII
- [10] Anatomy For Artists Eugene Wolff Fourth Edition (1962)

Copyright (C) Fotis Chatzinikos 1999

Appendix IV – Bibliography

- [11] **OpenGL, GLU and GLUT specification manuals**
- [12] <u>http://www.opengl.org</u>
- [13] <u>http://www.gamasutra.com</u>
- [14] many more Internet sites

DrawSprocket & OpenGL Tutorial for CodeWarrior

by Morgan Aldridge

Intro

I'll admit it, I'm not the best Mac developer in the world. So why am I writing this tutorial? Because I love it. I've been cc from '040 68K Macs to a PowerPC G3 (iMac, Rev B). So what was the first thing I set out to learn on PowerPC? OpenGL forums and tutorials out there for us Mac developers so I had to teach myselft GLUT from the demos that came with App some help.

When I finally taught myself how to make a true Mac application (sorry, I don't consider GLUT applications to be TRUE others how to do it (better late than never). Why would I want to make a MacOS application when I could use GLUT? W the significant decrease in size that I could get by not using GLUT. One problem that had been bugging me about makin (with GLUT I can just recompile under Windows and have the exact program running perfectly), but by the time I had fit

What You'll Need To Get Started

I know, you're getting tired of listening to me, so on to the tutorial. Basically you should already know how to do MacOS out <u>Macintosh C</u>. You will also need <u>Apple's OpenGL SDK 1.0</u> and the <u>DrawSprocket SDK</u>. If you don't already know he which has tons of OpenGL tutorials (including MacOS ports). Oh, and don't forget a compiler, if you don't have one I su the Discovery Programming edition) or if you don't want to spend any money, or don't have any to spend, you can get <u>N</u> explained in this tutorial which is aimed toward CodeWarrior users.

Preparing Your Project for OpenGL

In order for your application to take advantage of OpenGL you will need to add the following OpenGL stub libraries to y compiler then refer to its user documentation):

- OpenGLLibraryStub
- OpenGLUtilityStub
- OpenGLMemoryStub

Then to gain access to DrawSprocket functions you need to add DrawSprocketLib. The application that I will be develop screenshot of the CodeWarrior Pro 5 project window for it:

	glDrawSprocl	ketTest.mcp			E
	Link Order Targets				
giD	PrawSprocketTest DEBUG 🛛 🚽	* * •			
*	File	Code	Data	•	*
⊽ (🐧 Sources	2K	498	: •)	
	👔 main.c	2784	498	٠	
)	🐧 Resources	0	0		
V (🐧 ANSI Libraries	108K	15K	•	
	🔚 MSL C.PPC.Lib	92796	13964	٠	
	🔁 MSL SIOUX.PPC.Lib	18556	2307	٠	
V (🐧 Mac Libraries	14K	3K	•	
	🔁 MSL RuntimePPC.Lib	14544	3780	٠	
	🔁 InterfaceLib	0	0	٠	
	🔁 MathLib	0	0	٠	
	🔚 DrawSprocketLib	0	0	•	
V 🖌 🚺	🐧 OpenGL Libraries 🛛 🤨	0	0	•	
	🔚 OpenGLLibraryStub	0	0	٠	
	🔀 OpenGLMemoryStub	0	0	٠	
	🔚 OpenGLUtilityStub	0	0	٠	
_	🔁 tk.lib	0	0	٠	
	11 files	125K	20K		4

(Note that I also added tk.lib, this is a toolkit library which I use for image loading on occasion)

Now On To The Good Stuff

Well, I'm just going to go straight down through my "main.c" file and explain stuff, so first things first, included header fi

/**> HEADER FILES <**/ #include <stdlib.h></stdlib.h>	// ANSI C cross platform headers
<pre>#include <stdio.h> #include <types.h></types.h></stdio.h></pre>	// Standard MacOS Headers
<pre>#include <memory.h> #include <quickdraw.h></quickdraw.h></memory.h></pre>	
<pre>#include <fonts.h> #include <events.h></events.h></fonts.h></pre>	
<pre>#include <menus.h></menus.h></pre>	
<pre>#include <windows.h></windows.h></pre>	
<pre>#include <textedit.h></textedit.h></pre>	
<pre>#include <dialogs.h> #include <osutils.h></osutils.h></dialogs.h></pre>	
<pre>#include <toolutils.h></toolutils.h></pre>	
<pre>#include <segload.h></segload.h></pre>	
<pre>#include <sound.h></sound.h></pre>	
<pre>#include <drawsprocket.h></drawsprocket.h></pre>	// DrawSprocket
<pre>#include <agl.h></agl.h></pre>	// Apple's OpenGL
<pre>#include <glu.h></glu.h></pre>	<pre>// Used for setting perspective and making objects</pre>
<pre>#include <tk.h></tk.h></pre>	<pre>// Used for loading images</pre>

You should already know the first two headers, they just provide standard, non-OS specific memory, file, and other func important new ones: DrawSprocket.h provides access to DrawSprocket functions; agl.h provides access to Apple's Ope the perspective when rendering and also provides easy functions for creating spheres, columns, disks, etc; tk.h is only u you don't have to include it.

Next we come to my constant declarations:

/**> CONSTANT DECLARATIONS <**/ #define kMoveToFront	(WindowPtr) - 1L
// Screen Dimensions #define SCREEN_WIDTH #define SCREEN_HEIGHT	640 480
<pre>// Texture filters #define NEAREST #define LINEAR #define MIPMAP</pre>	0 1 2

Most of these won't make sense until you see them when they are used, but I'll describe them now anyway. kMoveToFre the front window. SCREEN_WIDTH and SCREEN_HEIGHT are also used for creating the new window for specifying tl LINEAR, and MIPMAP specify the type filtering to use when loading a texture with a custom function.

Next up: global variables. Only the first three of these are specifically used for DrawSprocket and AGL, but I will explain a

/**> GLOBAL VARIABLES <**/		
DSpContextAttributes	gDSpContextAttributes;	// Global DrawSprocket context attri
DSpContextReference	gDSpContext;	// The global DrawSprocket context
AGLContext	gOpenGLContext;	// The global OpenGL (AGL) context
// Texture Maps		
GLuint	gTutorialTexture;	
// Lighting Info		
GLfloat	$gLightAmbient[] = \{ 0.5,$	
GLfloat	gLightDiffuse[] = { 1.0,	1.0, 1.0, 1.0 };

GLfloat	gLightPosition[] = { 0.5, 1.0, 2.0, 1.0 };
// Material Info	
GLfloat	gMaterialAmbient[] = { 0.5, 0.5, 0.5, 1.0 };
GLfloat	gMaterialDiffuse[] = { 0.5, 0.5, 0.5, 1.0 };
GLfloat	gMaterialSpecular[] = { 0.9, 0.9, 0.9, 1.0 };
GLfloat	gMaterialShininess = 25;

gDSpContextAttributes is used for storing the attributes of the DrawSprocket context after creating it. Attributes includ created with gDSpContextAttributes, it'll get used to return the screen to it's original resolution and bit depth when the at things such as color depth, depth buffer depth, and more, this will get explained more later. gLightAmbient, gLightDiffue lighting works because there are better tutorials out there which can teach you this). gMaterialAmbient, gMaterialDiffuse also used for the lighting, but describes the obejects not the lights themselves.

You know we're starting to get to the good stuff when you reach the function prototypes and that's where we are now. Y to write a tutorial, but there are those out there which might want the extra explenation. Well, the function prototypes are really essential and which aren't.

```
/**> FUNCTION PROTOTYPES <**/
void
               ToolboxInit( void );
CGrafPtr SetupScreen( void );
void
               CreateWindow( CGrafPtr &theFrontBuffer );
void
                ShutdownScreen( CGrafPtr theFrontBuffer );
AGLContext
              SetupAGL( AGLDrawable window );
               CleanupAGL( AGLContext context );
void
               Reshape3D( int w, int h );
void
               InitGL( void );
void
void
               DrawGL( AGLContext context );
void
                LoadGLTexture( char *fileName, GLuint *texture, int filter );
```

ToolboxInit() initializes the MacOS Toolbox for your application (this gives you application the ability to draw windows screen, creates a new DrawSprocket context, creates a new window, and then fades back in. CreateWindow() is called free application quits and returns the screen to its original state. SetupAGL() creates a new AGL context which is used by Or and the like. CleanupAGL() gets rid of an AGL context. InitGL() initializes OpenGL things such as depth testing, backfac DrawGL() does all the OpenGL drawing and swaps the offscreen buffer at the end. LoadGLTexture() is a function which l

The main() Function

```
/****************** main() <*****/
void main( void )
{
     CGrafPtr theScreen;
     // Do a bunch of MacOS Inits
     ToolboxInit();</pre>
```

The call to ToolboxInit() will prepare our application to work as a MacOS application, if we don't call it our application w not be able take advantage of the MacOS GUI at all.

```
// Prepare the screen
HideCursor();
theScreen = SetupScreen();
```

First we'll hide the cursor, then we will prepare the screen using DrawSprocket by calling SetupScreen(). SetupScreen() r application.

```
// Setup the OpenGL context
gOpenGLContext = SetupAGL( ( AGLDrawable )theScreen );
```

SetupAGL() returns an AGL context for us to use, but if it doesn't create it correctly then the application will just quit. We

```
// Init OpenGL settings
InitGL();
```

InitGL() will load our textures, set up depth testing, lighting, and backface culling.

Here we will keep drawing our OpenGL scene until the mouse button is pressed. This is where you would normally put a this is supposed to be a really simple tutorial so I didn't bother with it.

```
// Get rid of the texture I loaded
glDeleteTextures( 1, &gTutorialTexture );
```

Here I'm deleting the texture that was created during InitGL() because the application is finished running (the mouse but

```
// Clean up the stuff we set up
CleanupAGL( gOpenGLContext );
ShutdownScreen( theScreen );
ShowCursor();
```

CleanupAGL() will dispose of our AGL context for us (so don't try to use it after this) and ShutdownScreen() will get rid and color depth. Then we have to call ShowCursor(), remember we hid it at the beginning of the program, so that we don

```
FlushEvents( everyEvent, 0 );
ExitToShell();
```

}

Now we clear all events from the event que so that other applications don't get any events that were supposed to go to ou but since Apple has been puting it there in a bunch of example applications I figured I might as well do it too.

The ToolboxInit() Function

There's really not much to this function, all it does is call a bunch of MacOS functions which let it draw to the screen, us

you need to know about it beyond that. Most tutorials don't include this function, they leave it for the developer to figure many, many, years ago, so I included it anyway.

The SetupScreen() Function

This makes a call to DSpStartup() which is a DrawSprocket function which registers our application with DrawSprocket

```
// Set the Context Attributes
gDSpContextAttributes.displayWidth = SCREEN_WIDTH;
gDSpContextAttributes.displayHeight = SCREEN_HEIGHT;
gDSpContextAttributes.colorNeeds = kDSpColorNeeds_Require;
gDSpContextAttributes.displayDepthMask = kDSpDepthMask_16;
gDSpContextAttributes.displayBestDepth = 16;
gDSpContextAttributes.pageCount = 1;
```

This is where we set the attributes that we want our screen to have, they are not necissarily what we will get, for example that resolution so it will give use 640x480 with a 512x384 window in it. But, we can also cause some problems, lets say we we'll have problems and won't be able to create a context at all.

DSpFindBestContext() tries to find a context that closely matches the attributes that we pass to it.

DSpContext_Reserve() reserves the context for our application.

This fades the screen out to black. Note that we're not telling it what context to fade out, so you can use this command w

Now we've set the screen to the state of the new context we created.

Fade the screen back in (works exactly like DSpContext_FadeGammaOut(), but in reverse).

```
// Create a window to draw in
CreateWindow( theFrontBuffer );
```

This is where we create a window. Apple usually puts the window code in this function, but I prefer to put in a seperate

```
return theFrontBuffer;
```

The last thing we do is return the color graphics port that CreateWindow() made for us. As you can see, the functions for DrawSprocket.h and try to figure out how to do other stuff if you want to.

The CreateWindow() Function

}

```
/***************** CreateWindow() <****/
void
               CreateWindow( CGrafPtr &theFrontBuffer )
{
        Rect
                                  rect;
        AuxWinHandle
                        awh;
        CTabHandle
                                  theColorTable;
        OSErr
                                  error;
        RGBColor
                        backColor = { 0xFFFF, 0xFFFF, 0xFFFF };
        RGBColor
                         foreColor = { 0x0000, 0x0000, 0x0000 };
        // Set the window rect
        rect.top = rect.left = 0;
        DSpContext_LocalToGlobal( gDSpContext, ( Point* )&rect );
        rect.right = rect.left + SCREEN_WIDTH;
        rect.bottom = rect.top + SCREEN_HEIGHT;
```

That code creates the rect for the new window, not too hard to understand.

```
// Create a new color window
theFrontBuffer = ( CGrafPtr )NewCWindow( NULL, &rect, "\p", 0, plainDBox, kMoveToF
```

NewCWindow() creates a new color window which we store in our color graphics port (theFrontBuffer), we pass in the r borders, title bar, or anything like that), and tell it to move it to the front (remember that from the constant declarations?)

```
// set the content color of the window to black to avoid a white flash when the wi
if ( GetAuxWin( ( WindowPtr )theFrontBuffer, &awh ) )
{
    theColorTable = ( **awh ).awCTable;
    error = HandToHand( ( Handle* )&theColorTable );
    if ( error )
        DebugStr( "\pOut of memory!" );
    ( **theColorTable ).ctTable[wContentColor].rgb.red = 0;
    ( **theColorTable ).ctTable[wContentColor].rgb.green = 0;
```

```
( **theColorTable ).ctTable[wContentColor].rgb.blue = 0;
CTabChanged( theColorTable );
// the color table will be disposed by the window manager when the window
SetWinColor( ( WindowPtr )theFrontBuffer, ( WCTabHandle )theColorTable );
}
```

All that code does is create a color table for the window and change the content color of it to black so that when the win black screen (we don't want that, we want it to look professional).

```
// Show the window
ShowWindow( ( GrafPtr )theFrontBuffer );
SetPort( ( GrafPtr )theFrontBuffer );
```

If you haven't learned ShowWindow() and ShowPort() yet you should. When a window is created it isn't visible, so you l that window the current graphics port (the current area to draw into).

```
// Set current pen colors
RGBForeColor( &foreColor );
RGBBackColor( &backColor );
```

To finish everything up we just set the forground drawing color to that listed at the top of the function (black) and the b

The ShutdownScreen() Function

}

Very little code to this function. We start out by calling DSpContext_FadeGammaOut() to fade out the screen, next we di (). We call DSpContext_SetState() to make our DrawSprocket context inactive, fade back in with DSpContext_FadeGamm DSpShutdown() to unregister our application from DrawSprocket, this will mean that our application can't use DrawSprc the process of quitting.

The SetupAGL() Function

```
/***********> SetupAGL() <*****/
AGLContext SetupAGL( AGLDrawable window )
{
    GLint attrib[] = { AGL_RGBA, AGL_DEPTH_SIZE, 24, AGL_DOUBLEBUFFER, AGL_NO
    AGLPixelFormat format;
    AGLContext context;
    GLboolean ok;

    // Choose an rgb pixel format
    format = aglChoosePixelFormat( NULL, 0, attrib );
    if ( format == NULL )
        return NULL;
}
</pre>
```

To create a picel format we pass aglChoosePixelFormat() our attributes (much like we did when creating our DrawSprocke

We use aglCreateContext() to create our AGL context from our pixel format that we just chose.

aglSetDrawable() sets a drawable (in this case a window) for the context. With a drawable set when you use OpenGL to r

And then you have to set the AGL context that we made as the current context. Notice that all along we've been making sused to this because one missed call can cause quite a few problems.

```
// The pixel format is no longer needed so get rid of it
aglDestroyPixelFormat( format );
return context;
```

To finish up this function we destroy our pixel format (we only needed it for creating our AGL context) and to return the 1

The CleanupAGL() Function

}

```
/************* CleanupAGL() <*****/
void CleanupAGL( AGLContext context )
{
     aglSetCurrentContext( NULL );
     aglSetDrawable( context, NULL );
     aglDestroyContext( context );
}</pre>
```

This is another one of those short functions, I guess it's just easier to throw stuff away than it is to make it. Anyway, We context to none, and finally destroy our context.

The Reshape3D() Function

```
glLoadIdentity();
glTranslatef( 0.0, 0.0, 0.0 );
```

This function can be used for any OpenGL application on any platform because it uses only standard OpenGL and GLU an identity matrix, sets the perspective with gluPerspective().

The InitGL() Function

}

}

```
/************> InitGL() <****/
void InitGL( void )
{
     // Load some textures
     LoadGLTexture( "Tutorial.sgi", &gTutorialTexture, LINEAR );
     glShadeModel( GL_SMOOTH );
     glEnable( GL_TEXTURE_2D );
</pre>
```

I said I wouldn't explain all the stuff in InitGL() very extensively because there are better places to learn it, but I will give loading the actual image, it leaves that upto tkRGBImageLoad(), or whatever image loader I am using at the time, but it d NEAREST, LINEAR, or MIPMAP (mipmap isn't really a filter, but is handy to have a quick method to load mipmaps).

```
// Enable depth testing
glClearDepth( 1.0 );
glDepthFunc( GL_LESS );
glEnable( GL_DEPTH_TEST );
```

Depth testing handles when polygons are infront or behind each other, so this code just sets a depth test function and ϵ

```
// Enable backface culling
glFrontFace( GL_CCW );
glCullFace( GL_BACK );
glEnable( GL_CULL_FACE );
```

glFrontFace() specifies which order the vertices are in for a polygon to be facing forward (clockwise or counter-clockwise)

```
// Configure a light
glLightfv( GL_LIGHT0, GL_AMBIENT, gLightAmbient );
glLightfv( GL_LIGHT0, GL_DIFFUSE, gLightDiffuse );
glLightfv( GL_LIGHT0, GL_POSITION, gLightPosition );
glLightModelf( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );
glEnable( GL_COLOR_MATERIAL );
glEnable( GL_LIGHT0 );
glEnable( GL_LIGHTING );
```

All these settings are for lighting, you should pay attention to GL_COLOR_MATERIAL, it lets materials have lighting at in handy).

```
// Setup Material stuff
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, gMaterialAmbient );
glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, gMaterialDiffuse );
glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, gMaterialSpecular );
glMaterialf( GL_FRONT_AND_BACK, GL_SHININESS, gMaterialShininess );
```

This is all material info (lighting mostly). You can put pretty much anything in the InitGL() function, it's another one of the

The DrawGL() Function

```
/************ DrawGL() <****/
void DrawGL( AGLContext context )
{
    static float rot;
    rot += 0.1;</pre>
```

I just continue adding to a rotation (by the way, all this simple application does is rotate a rectangle which has a different

```
// Clear color buffer to black
glClearColor( 0, 0, 0, 1 );
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
// Load the identity matrix
glLoadIdentity();
```

No fancy AGL stuff here, it's just a simple call to OpenGL's glClear() function which clears, in this case, the offscreen but

```
// Move the view back a bit
glTranslatef( 0, 0, -3 );
// Set the light's position
glLightfv( GL_LIGHT0, GL_POSITION, gLightPosition );
```

I start by calling glTranslatef() to move our scene back by three units so we're not sitting with our eyeballs right at the ol this every time you update, but right after you do camera translations and rotations so that the light doesn't appear to m

```
// Rotation for the quads
glRotatef( rot, 0, 1, 0 );
```

This just rotates the object.

```
// Draw the quads
glBindTexture( GL_TEXTURE_2D, gTutorialTexture );
glBegin( GL_QUADS );
         // Front Quad
         glNormal3f( 0, 0, 1 );
         glTexCoord2f( 1, 1 ); glVertex3f( 1, 0.5, 0 );
         glTexCoord2f( 0, 1 ); glVertex3f( -1, 0.5, 0 );
         glTexCoord2f( 0, 0.5 ); glVertex3f( -1, -0.5, 0 );
         glTexCoord2d( 1, 0.5 ); glVertex3f( 1, -0.5, 0 );
         // Back Quad
         glNormal3f( 0, 0, -1 );
         glTexCoord2f( 1, 0.5 ); glVertex3f( -1, 0.5, 0 );
         glTexCoord2f( 0, 0.5 ); glVertex3f( 1, 0.5, 0 );
         glTexCoord2f( 0, 0 ); glVertex3f( 1, -0.5, 0 );
         glTexCoord2d( 1, 0 ); glVertex3f( -1, -0.5, 0 );
glEnd();
```

The object is two texture mapped quads (one for each side of the rectangle) so I start by binding the texture (the top half specifying the texture coordinates and vertices that make up the quads.

```
// Copy the offscreen buffer to the screen
aglSwapBuffers( context );
```

}

This is the only line of code in the whole function that is specific to AGL, aglSwapBuffers() does the same thing as glutS¹ drawable that we set for out AGL context.

The LoadGLTexture() Function

This function is only needed for this tutorial, it's not one of the DrawSprocket or AGL functions needed, but I do find it f

```
/************> LoadGLTexture() <****/
void LoadGLTexture( char *fileName, GLuint *texture, int filter )
{
    TK_RGBImageRec *tempTexture;
    // Load the file
    tempTexture = tkRGBImageLoad( fileName );</pre>
```

the tkRGBImageLoad() function is part of tk.lib, it loads and RGB image, in this case an sgi image in RGB format with RL

```
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
// Generate a texture
glGenTextures( 1, texture );
glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
```

glGenTextures() allocates memory for a texture. For this function I specify that I am only generating one texture, but one memory for more than one texture and pass in an array to store the addresses in.

```
switch ( filter )
{
         case NEAREST:
                  // Create Nearest Filtered Texture
                  glBindTexture( GL_TEXTURE_2D, *texture );
                  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEARES
                  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEARES
                  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, tempTexture->sizeX, temp
                  break;
         case LINEAR:
                  // Create Linear Filtered Texture
                  glBindTexture( GL_TEXTURE_2D, *texture );
                  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAF
                  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAF
                  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, tempTexture->sizeX, temp
                  break;
         case MTPMAP:
                  // Create MipMapped Texture
                  glBindTexture( GL_TEXTURE_2D, *texture );
                  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAF
                  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAF
                  gluBuild2DMipmaps( GL_TEXTURE_2D, GL_RGB, tempTexture->sizeX, te
                  break;
}
```

Each case in the switch statement takes tempTexture and sets a filter (GL_NEAREST or GL_LINEAR) and then calls gIT MIPMAP, which is more than just a filter, it actally generates optimized textures for various distances so they look better

```
free( tempTexture );
```

}

Now we just get rid of tempTexture because we've already copied it's data into our new texture. Remember to use glDelet () because if you don't they will stay in your 3D graphics accelerator card's memory until the computer is shut down, and times.

Let's Wrap Things Up (a.k.a. Where's The Download?)

Well, that was the last function, so the tutorial is officially done. The source code and CodeWarrior Pro 5 project for glD: any questions, comments, or corrections feel free to e-mail me at classic@sover.net. I hope you found this tutorial useful

The OpenGL[®] Graphics System: A Specification (Version 1.2.1)

Mark Segal Kurt Akeley

Editor (version 1.1): Chris Frazier Editor (versions 1.2, 1.2.1): Jon Leech

Copyright © 1992-1999 Silicon Graphics, Inc.

This document contains unpublished information of Silicon Graphics, Inc.

This document is protected by copyright, and contains information proprietary to Silicon Graphics, Inc. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of Silicon Graphics, Inc. is strictly prohibited. The receipt or possession of this document does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

U.S. Government Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227.7013 and/or in similar or successor clauses in the FAR or the DOD or NASA FAR Supplement. Unpublished rights reserved under the copyright laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

OpenGL is a registered trademark of Silicon Graphics, Inc. Unix is a registered trademark of The Open Group. The "X" device and X Windows System are trademarks of The Open Group.

Contents

1	\mathbf{Intr}	oduction	1
	1.1	Formatting of Optional Features	1
	1.2	What is the OpenGL Graphics System?	1
	1.3	Programmer's View of OpenGL	2
	1.4	Implementor's View of OpenGL	2
	1.5	Our View	3
2	Ope	enGL Operation	4
	2.1	OpenGL Fundamentals	4
		2.1.1 Floating-Point Computation	6
	2.2	GL State	6
	2.3	GL Command Syntax	7
	2.4	Basic GL Operation	9
	2.5	GL Errors	11
	2.6	$\operatorname{Begin}/\operatorname{End}\operatorname{Paradigm}.............$	12
		2.6.1 Begin and End Objects	15
		2.6.2 Polygon Edges	18
		2.6.3 GL Commands within Begin / End	19
	2.7	Vertex Specification	19
	2.8	Vertex Arrays	21
	2.9	Rectangles	28
	2.10	Coordinate Transformations	28
		2.10.1 Controlling the Viewport	30
		2.10.2 Matrices	31
		2.10.3 Normal Transformation	34
		2.10.4 Generating Texture Coordinates	36
	2.11	Clipping	38
	2.12	Current Raster Position	40
	2.13	Colors and Coloring	43

CONTENTS

		2.13.1	Lighting
		2.13.2	Lighting Parameter Specification
		2.13.3	ColorMaterial 51
		2.13.4	Lighting State
		2.13.5	Color Index Lighting
		2.13.6	Clamping or Masking 54
		2.13.7	Flatshading
		2.13.8	Color and Texture Coordinate Clipping 55
		2.13.9	Final Color Processing
3	Ras	terizat	ion 57
	3.1	Invaria	unce
	3.2		asing
	3.3	Points	
		3.3.1	Point Rasterization State
	3.4		$egments \dots \dots$
		3.4.1	Basic Line Segment Rasterization
		3.4.2	Other Line Segment Features
		3.4.3	Line Rasterization State
	3.5		ns
		3.5.1	Basic Polygon Rasterization
		3.5.2	Stippling
		3.5.3	Antialiasing
		3.5.4	Options Controlling Polygon Rasterization 73
		3.5.5	Depth Offset
		3.5.6	Polygon Rasterization State
	3.6		Rectangles
		3.6.1	Pixel Storage Modes
		3.6.2	The Imaging Subset
		3.6.3	Pixel Transfer Modes
		3.6.4	Rasterization of Pixel Rectangles
		3.6.5	Pixel Transfer Operations
	3.7	Bitma	ps
	3.8		ing
		3.8.1	Texture Image Specification
		3.8.2	Alternate Texture Image Specification Commands 118
		3.8.3	Texture Parameters
		3.8.4	Texture Wrap Modes
		3.8.5	Texture Minification
		3.8.6	Texture Magnification

		3.8.7	Texture State and Proxy State			•		131
		3.8.8 '	Texture Objects					132
		3.8.9 ′	Texture Environments and Texture Functions					135
		3.8.10 '	Texture Application					138
	3.9	Color S	um					138
	3.10	Fog						138
	3.11	Antialia	sing Application \ldots \ldots \ldots \ldots \ldots \ldots	•		•		140
4	Por-	Fragme	ent Operations and the Framebuffer					141
-	4.1		gment Operations					
	1.1		Pixel Ownership Test					
			Scissor test					
			Alpha test					
			Stencil test \ldots \ldots \ldots \ldots \ldots \ldots					
			Depth buffer test					
			Blending					
			Dithering					
			Logical Operation					
	4.2		Framebuffer Operations					
			Selecting a Buffer for Writing					
			Fine Control of Buffer Updates					
			Clearing the Buffers					
		4.2.4 '	The Accumulation Buffer					155
	4.3	Drawing	g, Reading, and Copying Pixels					156
		4.3.1	Writing to the Stencil Buffer					156
		4.3.2	Reading Pixels					156
		4.3.3	Copying Pixels					162
		4.3.4	$Pixel Draw/Read state \dots \dots \dots \dots \dots \dots$	•		•	• •	162
5	Spe	cial Fun	octions					164
	5.1		ors					
	5.2		n					
	5.3		\mathbf{k}					
	5.4		Lists					
	5.5	1 0	nd Finish					
	5.6							
6	Stat	e and S	state Requests					181
5	6.1		$\operatorname{g} \operatorname{GL} \operatorname{State} \ldots \ldots$					
	0.1	• •	Simple Queries					
		0.1.1		·	•	•	• •	TOT

CONTENTS

		6.1.2 l	Data Conv	ersions		• •	•					•		•	•	• •	•	•	182
		6.1.3 I	Enumerate	d Queri	ies .											• •	•		182
		6.1.4	Texture Q	ueries												• •	•		184
		6.1.5	Stipple Qu	ery												• •	•		185
		6.1.6	Color Mat:	rix Que	ry.												•		185
		6.1.7 (Color Tabl	e Query	7												•		185
		6.1.8	Convolutio	n Quer	у						•				•	• •	•		186
		6.1.9 I	Histogram	Query													•		187
		6.1.10 I	Minmax Q	uery .							•				•	• •	•		188
		6.1.11 I	Pointer an	d String	g Qué	erie	s					•			•	• •	•		189
		6.1.12 \$	Saving and	Restor	ing S	Stat	е				•				•	• •	•		189
	6.2	State Ta	ables						•			•			•	• •	•		193
	-																		
Α		riance																	218
	A.1	-	ability																
	A.2	-	ass Algorit																
	A.3		ice Rules																
	A.4	What A	ll This Me	eans .	•••	• •	·	• •	·	• •	•	·	•••	·	•	• •	•	•	221
в	Core	ollaries																	222
D	001	onaries																	
_																			005
_	Vers	sion 1.1	A																225
_	Vers C.1	sion 1.1 Vertex A	Array																225
_	Vers C.1 C.2	sion 1.1 Vertex A Polygon	. Offset												•		•	•	$\begin{array}{c} 225\\ 226 \end{array}$
_	Vers C.1 C.2 C.3	sion 1.1 Vertex A Polygon Logical	Offset . Operation		· · ·	 	•	•••	•	•••	•	•	•••	•	•	• •	•	•	225 226 226
_	Vers C.1 C.2 C.3 C.4	sion 1.1 Vertex A Polygon Logical Texture	Offset . Operation Image Fo:	\cdots	· · · ·	· · ·	•	 		· ·	•		 	•	•	• •	•	•	225 226 226 226
_	Vers C.1 C.2 C.3 C.4 C.5	sion 1.1 Vertex A Polygon Logical Texture Texture	Offset . Operation Image Fo Replace I	· · · · · · · · · rmats Environn	\cdots	 	•	· ·		· ·		•	••••	•	•	• •	• •		225 226 226 226 226
_	Vers C.1 C.2 C.3 C.4 C.5 C.6	sion 1.1 Vertex A Polygon Logical Texture Texture Texture	Offset . Operation Image For Replace B Proxies .	· · · · · rmats Environn	\cdots	· · · · · · · · · · · · · · · · · · ·		· · ·		· · · · · · · · · · · · · · · · · · ·		•	 	•	•	• •	• •	• • •	225 226 226 226 226 226 227
_	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7	sion 1.1 Vertex A Polygon Logical Texture Texture Texture Copy Te	Offset . Operation Image Fo: Replace I Proxies . exture and	 rmats Environn Subtex	 	· · · · · · · · · · · · · · · · · · ·	• • •	· · · · · · · · · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·			· · · · · ·			· · ·	· •		225 226 226 226 226 227 227
_	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Te Texture	Offset . Operation Image For Replace H Proxies . exture and Objects	rmats Invironi Subtex	 ment 	· · · · · · · · · · · · · · · · · · ·	• • • •	· · · · · · · · · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·		•	· · · · · ·			· · ·	· • •		225 226 226 226 226 227 227 227
_	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Texture Other C	Offset . Operation Image Fo: Replace H Proxies . exture and Objects Changes .	rmats Environi Subtex 	 	· · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·	· · · ·		· · · · · · · · ·		· · · · · · · · ·	· · ·	· • • • • • • • • • • • • • • • • • • •		225 226 226 226 227 227 227 227 227
_	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Texture Other C	Offset . Operation Image For Replace H Proxies . exture and Objects	rmats Environi Subtex 	 	· · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·	· · · ·		· · · · · · · · ·		· · · · · · · · ·	· · ·	· • • • • • • • • • • • • • • • • • • •		225 226 226 226 227 227 227 227 227
C	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9 C.10	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Texture Other C	Offset . Operation Image For Replace H Proxies . exture and Objects Changes . ledgement	rmats Environi Subtex 	 	· · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·		· · · · · · · · · · · · · · · · · · ·	· · ·		· · · · · · · · ·		· · · · · · · · ·	· · ·	· • • • • • • • • • • • • • • • • • • •	· · · ·	225 226 226 226 227 227 227 227 227
C	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9 C.10	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Te Texture Other Co Acknow	Offset . Operation Image For Replace H Proxies . exture and Objects Changes . ledgement	rmats Environi Subtex	 ment 	 . .<	· · ·	· · · · · · · · ·	· · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	• • • • •	· · · · · · · · · · · · · · · · · · ·		• • • • • •	· · ·		· · · ·	225 226 226 226 227 227 227 227 227 228
C	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9 C.10 Vers	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Te Texture Other C Acknow Sion 1.2	Offset . Operation Image Fo: Replace H Proxies . exture and Objects Changes . ledgement	$\begin{array}{c} \dots \\ \dots $	 ment 	 . .<	· · · ·	· · · · · · · · · · · · · · · · · · ·	· · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · ·	· · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · ·	· · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· • • • • • • • • • • • • • • • • • • •	· · · ·	 225 226 226 227 227 227 227 228 230
C	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9 C.10 Vers D.1	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Te Texture Other C Acknow sion 1.2 BGRA	Offset . Operation Image Fo: Replace H Proxies . exture and Objects Changes . ledgement	rmats Environn Subtex s s l Textu	 ment 	 . .<	· · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · ·	· · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · ·	· · · · · · · · · · · · · · · · · · ·	· · ·		· · · · ·	225 226 226 227 227 227 227 227 228 230
C	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9 C.10 Vers D.1 D.2	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Te Texture Other C Acknow sion 1.2 Three-D BGRA	Offset . Operation Image Fo: Replace H Proxies . exture and Objects Changes . ledgement Dimensiona Pixel Form Pixel Form	rmats Invironi Subtex Subtex s s al Textu		 . .<	· · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · ·	· · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · ·	· · · · ·	· · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·		· · · · ·	225 226 226 227 227 227 227 227 227 228 230 230
C	Vers C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9 C.10 Vers D.1 D.2 D.3	sion 1.1 Vertex A Polygon Logical Texture Texture Copy Te Texture Other C Acknow Sion 1.2 Three-D BGRA Packed Normal	Offset . Operation Image Fo: Replace H Proxies . exture and Objects Changes . ledgement Dimensiona Pixel Form	rmats Environn Subtex Subtex S s s l Textu nats nats		 . .<	· · · · ·	· · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · · ·	· · · · · · · · · · · · · · · · · ·	· · · · ·		· · · · · · · · · · · · · · · · · · ·		· · · · ·	225 226 226 227 227 227 227 227 228 230 230 230 230

	D.7	Textur	e Level of Detail Control
	D.8	Vertex	Array Draw Element Range
	D.9	Imagin	g Subset
		D.9.1	Color Tables
		D.9.2	Convolution
		D.9.3	Color Matrix
		D.9.4	Pixel Pipeline Statistics
		D.9.5	Constant Blend Color
		D.9.6	New Blending Equations
	D.10	Acknow	wledgements
\mathbf{E}	Vers	sion 1.2	2.1 238
\mathbf{F}	AR	B Exte	nsions 239
	F.1	Namin	g Conventions
	F.2	Multit	$exture \dots \dots \dots \dots \dots \dots \dots \dots \dots $
		F.2.1	Dependencies
		F.2.2	Issues
		F.2.3	Changes to Section 2.6 (Begin/End Paradigm) 240
		F.2.4	Changes to Section 2.7 (Vertex Specification) 241
		F.2.5	Changes to Section 2.8 (Vertex Arrays) 243
		F.2.6	Changes to Section 2.10.2 (Matrices)
		F.2.7	Changes to Section 2.10.4 (Generating Texture Coor-
			dinates) $\ldots \ldots 245$
		F.2.8	Changes to Section 2.12 (Current Raster Position) 246
		F.2.9	Changes to Section 3.8 (Texturing)
		F.2.10	Changes to Section 3.8.5 (Texture Minification) 248
		F.2.11	Changes to Section 3.8.8 (Texture Objects) 248
		F.2.12	Changes to Section 3.8.10 (Texture Application) 249
		F.2.13	Changes to Section 5.1 (Evaluators)
		F.2.14	Changes to Section 5.3 (Feedback)
			Changes to Section 6.1.2 (Data Conversions) 251
		F.2.16	Changes to Section 6.1.12 (Saving and Restoring State)251

Index of OpenGL Commands

 $\mathbf{256}$

v

List of Figures

2.1	Block diagram of the GL.
2.2	Creation of a processed vertex from a transformed vertex and
	current values
2.3	Primitive assembly and processing
2.4	Triangle strips, fans, and independent triangles 16
2.5	Quadrilateral strips and independent quadrilaterals 17
2.6	Vertex transformation sequence
2.7	Current raster position
2.8	Processing of RGBA colors
2.9	Processing of color indices
2.10	ColorMaterial operation
3.1	Rasterization
3.2	Rasterization of non-antialiased wide points
3.3	Rasterization of antialiased wide points
3.4	Visualization of Bresenham's algorithm
3.5	Rasterization of non-antialiased wide lines
3.6	The region used in rasterizing an antialiased line segment 69
3.7	Operation of DrawPixels
3.8	Selecting a subimage from an image
3.9	A bitmap and its associated parameters
3.10	A texture image and the coordinates used to access it 118 $$
4.1	Per-fragment operations
4.2	Operation of ReadPixels
4.3	Operation of CopyPixels
5.1	Map Evaluation
5.2	Feedback syntax

LIST OF FIGURES

F.1	Creation of a processed vertex from a transformed vertex and	
	current values. $\ldots \ldots 2$	41
F.2	Current raster position	46
F.3	Multitexture pipeline	49

List of Tables

2.1	GL command suffixes	8
2.2	GL data types	10
2.3	Summary of GL errors	13
2.4	Vertex array sizes (values per vertex) and data types	22
2.5	Variables that direct the execution of InterleavedArrays.	26
2.6	Component conversions	44
2.7	Summary of lighting parameters.	46
2.8	Correspondence of lighting parameter symbols to names	50
2.9	Polygon flatshading color selection	55
3.1	PixelStore parameters pertaining to one or more of Draw -	
	$\mathbf{Pixels}, \ \mathbf{TexImage1D}, \ \mathbf{TexImage2D}, \ \mathbf{and} \ \mathbf{TexImage3D}. \ .$	76
3.2	PixelTransfer parameters	78
3.3	PixelMap parameters.	79
3.4	Color table names	80
3.5	DrawPixels and ReadPixels types	91
3.6	DrawPixels and ReadPixels formats	92
3.7	Swap Bytes Bit ordering.	92
3.8	Packed pixel formats	94
3.9	UNSIGNED_BYTE formats. Bit numbers are indicated for each	
	component	95
	UNSIGNED_SHORT formats	96
	UNSIGNED_INT formats	97
	Packed pixel field assignments	98
	1	103
3.14	Computation of filtered color components	104
3.15	Conversion from RGBA pixel components to internal texture,	
	, 1	114
3.16	Correspondence of sized internal formats to base internal for-	
	mats	115

3.17	Texture parameters and their values	4
3.18	Replace and modulate texture functions	6
3.19	Decal and blend texture functions	7
4 1		
4.1	Values controlling the source blending function and the source $(A + 1 + A)$	0
4.0	blending values they compute. $f = \min(A_s, 1 - A_d)$ 146	ð
4.2	Values controlling the destination blending function and the	0
4.0	destination blending values they compute	
4.3	Arguments to LogicOp and their corresponding operations. 15	
4.4	Arguments to DrawBuffer and the buffers that they indicate 15:	2
4.5	PixelStore parameters pertaining to ReadPixels,	
	GetTexImage1D, GetTexImage2D, GetTexImage3D,	
	GetColorTable, GetConvolutionFilter, GetSeparable-	~
	Filter, GetHistogram, and GetMinmax	
4.6	ReadPixels index masks	0
4.7	ReadPixels GL Data Types and Reversed component con-	
	version formulas	1
5.1	Values specified by the <i>target</i> to Map1	5
5.2	Correspondence of feedback type to number of values per vertex.174	
6.1	Texture, table, and filter return values	
6.2	Attribute groups	1
	Attribute groups 19 State variable types 19	$\frac{1}{2}$
$6.2 \\ 6.3 \\ 6.4$	Attribute groups 19 State variable types 19 GL Internal begin-end state variables (inaccessible) 19	$1 \\ 2 \\ 4$
$\begin{array}{c} 6.2 \\ 6.3 \end{array}$	Attribute groups19State variable types19GL Internal begin-end state variables (inaccessible)19Current Values and Associated Data19	$1 \\ 2 \\ 4 \\ 5$
$6.2 \\ 6.3 \\ 6.4$	Attribute groups 19 State variable types 19 GL Internal begin-end state variables (inaccessible) 19	$1 \\ 2 \\ 4 \\ 5$
$6.2 \\ 6.3 \\ 6.4 \\ 6.5$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data196Transformation state197	$ 1 \\ 2 \\ 4 \\ 5 \\ 6 \\ 7 $
$6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197	1 2 4 5 6 7 8
$6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data198Vertex Array Data199Transformation state199Coloring198Lighting (see also Table 2.7 for defaults)198	1 2 4 5 6 7 8 9
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197Lighting (see also Table 2.7 for defaults)198Lighting (cont.)200	1 2 4 5 6 7 8 9 0
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data198Vertex Array Data199Transformation state199Coloring198Lighting (see also Table 2.7 for defaults)198	1 2 4 5 6 7 8 9 0
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \\ 6.10 \\ 6.11 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197Lighting (see also Table 2.7 for defaults)197Lighting (cont.)207Rasterization207Texture Objects207	$1 \\ 2 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 0 \\ 1$
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \\ 6.10 \\ 6.11 \end{array}$	Attribute groups19State variable types19GL Internal begin-end state variables (inaccessible)19Current Values and Associated Data19Vertex Array Data19Transformation state19Coloring19Lighting (see also Table 2.7 for defaults)19Lighting (cont.)20Rasterization20	$1 \\ 2 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 0 \\ 1 \\ 2$
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \\ 6.10 \\ 6.11 \\ 6.12 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197Lighting (see also Table 2.7 for defaults)197Lighting (cont.)207Rasterization207Texture Objects207	124567890123
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \\ 6.10 \\ 6.11 \\ 6.12 \\ 6.13 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197Lighting (see also Table 2.7 for defaults)198Lighting (cont.)200Rasterization200Texture Objects200Texture Objects (cont.)200	1245678901234
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \\ 6.10 \\ 6.11 \\ 6.12 \\ 6.13 \\ 6.14 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197Lighting (see also Table 2.7 for defaults)197Lighting (cont.)207Texture Objects207Texture Objects (cont.)207Texture Environment and Generation207	12456789012345
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \\ 6.10 \\ 6.11 \\ 6.12 \\ 6.13 \\ 6.14 \\ 6.15 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197Lighting (see also Table 2.7 for defaults)197Lighting (cont.)207Rasterization207Texture Objects207Texture Objects (cont.)207Texture Environment and Generation207Framebuffer Control207Framebuffer Control207	124567890123456
$\begin{array}{c} 6.2 \\ 6.3 \\ 6.4 \\ 6.5 \\ 6.6 \\ 6.7 \\ 6.8 \\ 6.9 \\ 6.10 \\ 6.11 \\ 6.12 \\ 6.13 \\ 6.14 \\ 6.15 \\ 6.16 \\ 6.17 \end{array}$	Attribute groups197State variable types197GL Internal begin-end state variables (inaccessible)197Current Values and Associated Data197Vertex Array Data197Transformation state197Coloring197Lighting (see also Table 2.7 for defaults)197Lighting (cont.)207Rasterization207Texture Objects207Texture Objects (cont.)207Pixel Operations207Framebuffer Control207Framebuffer Control207	1245678901234567

LIST OF TABLES

6.20	Pixels (cont.)
6.21	Pixels (cont.)
6.22	Evaluators (GetMap takes a map name)
6.23	Hints
6.24	Implementation Dependent Values
6.25	More Implementation Dependent Values
6.26	Implementation Dependent Pixel Depths
6.27	Miscellaneous
F.1	Changes to State Tables
F.2	Changes to State Tables (cont.)
F.3	New State Introduced by Multitexture
F.4	New Implementation-Dependent Values Introduced by Mul-
	titexture

х

Chapter 1

Introduction

This document describes the OpenGL graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudimentary understanding of computer graphics. This means familiarity with the essentials of computer graphics algorithms as well as familiarity with basic graphics hardware and associated terms.

1.1 Formatting of Optional Features

Starting with version 1.2 of OpenGL, some features in the specification are considered optional; an OpenGL implementation may or may not choose to provide them (see section 3.6.2).

Portions of the specification which are optional are so labelled where they are defined. Additionally, those portions are typeset in gray, and state table entries which are optional are typeset against a gray background.

1.2 What is the OpenGL Graphics System?

OpenGL (for "Open Graphics Library") is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Most of OpenGL requires that the graphics hardware contain a framebuffer. Many OpenGL calls pertain to drawing objects such as points, lines, polygons, and bitmaps, but the way that some of this drawing occurs (such as when antialiasing or texturing is enabled) relies on the existence of a framebuffer. Further, some of OpenGL is specifically concerned with framebuffer manipulation.

1.3 Programmer's View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate a GL context and associate it with the window. Once a GL context is allocated, the programmer is free to issue OpenGL commands. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user's two- or three-dimensional model space to the two-dimensional screen. There are also calls to effect direct control of the framebuffer, such as reading and writing pixels.

1.4 Implementor's View of OpenGL

To the implementor, OpenGL is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable framebuffer, then OpenGL must be implemented almost entirely on the host CPU. More typically, the graphics hardware may comprise varying degrees of graphics acceleration, from a raster subsystem capable of rendering twodimensional lines and polygons to sophisticated floating-point processors capable of transforming and computing on geometric data. The OpenGL implementor's task is to provide the CPU software interface while dividing the work for each OpenGL command between the CPU and the graphics hardware. This division must be tailored to the available graphics hardware to obtain optimum performance in carrying out OpenGL calls.

OpenGL maintains a considerable amount of state information. This state controls how objects are drawn into the framebuffer. Some of this state is directly available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is drawn. One of the main goals of this specification is to make OpenGL state information explicit, to elucidate how it changes, and to indicate what its effects are.

1.5 Our View

We view OpenGL as a state machine that controls a set of specific drawing operations. This model should engender a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

Chapter 2

OpenGL Operation

2.1 **OpenGL Fundamentals**

OpenGL (henceforth, the "GL") is concerned only with rendering into a framebuffer (and reading values stored in that framebuffer). There is no support for other peripherals sometimes associated with graphics hardware, such as mice and keyboards. Programmers must rely on other mechanisms to obtain user input.

The GL draws *primitives* subject to a number of selectable modes. Each primitive is a point, line segment, polygon, or pixel rectangle. Each mode may be changed independently; the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer). Modes are set, primitives specified, and other GL operations described by sending *commands* in the form of function or procedure calls.

Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an endpoint of an edge, or a corner of a polygon where two edges meet. Data (consisting of positional coordinates, colors, normals, and texture coordinates) are associated with a vertex and each vertex is processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be *clipped* so that the indicated primitive fits within a specified region; in this case vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before the effects of a command are realized. This means, for example, that one primitive must be drawn completely before any subsequent one can affect the framebuffer. It also means that queries and pixel read operations return state consistent with complete execution of all previously invoked GL commands. In general, the effects of a GL command on either GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

In the GL, data binding occurs on call. This means that data passed to a command are interpreted when that command is received. Even if the command requires a pointer to data, those data are interpreted when the call is made, and any subsequent changes to the data have no effect on the GL (unless the same pointer is used in a subsequent command).

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. It does not provide a means for describing or modeling complex geometric objects. Another way to describe this situation is to say that the GL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves.

The model for interpretation of GL commands is client-server. That is, a program (the client) issues commands, and these commands are interpreted and processed by the GL (the server). The server may or may not operate on the same computer as the client. In this sense, the GL is "network-transparent." A server may maintain a number of GL contexts, each of which is an encapsulation of current GL state. A client may choose to connect to any one of these contexts. Issuing GL commands when the program is not connected to a context results in undefined behavior.

The effects of GL commands on the framebuffer are ultimately controlled by the window system that allocates framebuffer resources. It is the window system that determines which portions of the framebuffer the GL may access at any given time and that communicates to the GL how those portions are structured. Therefore, there are no GL commands to configure the framebuffer or initialize the GL. Similarly, display of framebuffer contents on a CRT monitor (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by the GL. Framebuffer configuration occurs outside of the GL in conjunction with the window system; the initialization of a GL context occurs when the window system allocates a window for GL rendering.

The GL is designed to be run on a range of graphics platforms with varying graphics capabilities and performance. To accommodate this variety, we specify ideal behavior instead of actual behavior for certain GL operations. In cases where deviation from the ideal is allowed, we also specify the rules that an implementation must obey if it is to approximate the ideal behavior usefully. This allowed variation in GL behavior implies that two distinct GL implementations may not agree pixel for pixel when presented with the same input even when run on identical framebuffer configurations.

Finally, command names, constants, and types are prefixed in the GL (by gl, GL_, and GL, respectively in C) to reduce name clashes with other packages. The prefixes are omitted in this document for clarity.

2.1.1 Floating-Point Computation

The GL must perform a number of floating-point operations during the course of its operation. We do not specify how floating-point numbers are to be represented or how operations on them are to be performed. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5 . The maximum representable magnitude of a floating-point number used to represent positional or normal coordinates must be at least 2^{32} ; the maximum representable magnitude for colors or texture coordinates must be at least 2^{10} . The maximum representable magnitude for all other floating-point values must be at least 2^{32} . $x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN x. $1 \cdot x = x \cdot 1 = x$. x + 0 = 0 + x = x. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to GL interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results.

Some calculations require division. In such cases (including implied divisions required by vector normalizations), a division by zero produces an unspecified result but must not lead to GL interruption or termination.

2.2 GL State

The GL maintains considerable state. This document enumerates each state variable and describes how each variable can be changed. For purposes of discussion, state variables are categorized somewhat arbitrarily by their function. Although we describe the operations that the GL performs on the framebuffer, the framebuffer is not a part of GL state.

We distinguish two types of state. The first type of state, called GL *server state*, resides in the GL server. The majority of GL state falls into this category. The second type of state, called GL *client state*, resides in the GL client. Unless otherwise specified, all state referred to in this document is GL server state; GL client state is specifically identified. Each instance of a GL context implies one complete set of GL server state; each connection from a client to a server implies a set of both GL client state and GL server state.

While an implementation of the GL may be hardware dependent, this discussion is independent of the specific hardware on which a GL is implemented. We are therefore concerned with the state of graphics hardware only when it corresponds precisely to GL state.

2.3 GL Command Syntax

GL commands are functions or procedures. Various groups of commands perform the same operation but differ in how arguments are supplied to them. To conveniently accommodate this variation, we adopt a notation for describing commands and their arguments.

GL commands are formed from a *name* followed, depending on the particular command, by up to 4 characters. The first character indicates the number of values of the indicated type that must be presented to the command. The second character or character pair indicates the specific type of the arguments: 8-bit integer, 16-bit integer, 32-bit integer, single-precision floating-point, or double-precision floating-point. The final character, if present, is v, indicating that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments. Two specific examples come from the **Vertex** command:

void Vertex3f(float x, float y, float z);

 and

void Vertex2sv(short v/2);

These examples show the ANSI C declarations for these commands. In general, a command declaration has the form¹

¹The declarations shown in this document apply to ANSI C. Languages such as C++

Letter	Corresponding GL Type
b	byte
s	short
i	int
f	float
d	double
ub	ubyte
us	ushort
ui	uint

Table 2.1: Correspondence of command suffix letters to GL argument types. Refer to Table 2.2 for definitions of the GL types.

```
 \begin{array}{l} rtype \ \mathbf{Name}\{\epsilon \mathbf{1234}\}\{\epsilon \ \mathbf{b} \ \mathbf{s} \ \mathbf{i} \ \mathbf{f} \ \mathbf{d} \ \mathbf{ub} \ \mathbf{us} \ \mathbf{ui}\}\{\epsilon \mathbf{v}\} \\ ( \ [args \ ,] \ T \ arg1 \ , \ \ldots \ , \ T \ argN \ [, \ args] \ ); \end{array}
```

rtype is the return type of the function. The braces $(\{\})$ enclose a series of characters (or character pairs) of which one is selected. ϵ indicates no character. The arguments enclosed in brackets ([args,]] and [, args]) may or may not be present. The N arguments arg1 through argN have type T, which corresponds to one of the type letters or letter pairs as indicated in Table 2.1 (if there are no letters, then the arguments' type is given explicitly). If the final character is not \mathbf{v} , then N is given by the digit 1, 2, 3, or 4 (if there is no digit, then the number of arguments is fixed). If the final character is \mathbf{v} , then only arg1 is present and it is an array of N values of the indicated type. Finally, we indicate an unsigned type by the shorthand of prepending a u to the beginning of the type name (so that, for instance, unsigned char is abbreviated uchar).

For example,

void Normal3{fd}(T arg);

indicates the two declarations

```
void Normal3f( float arg1, float arg2, float arg3 );
void Normal3d( double arg1, double arg2, double arg3 );
```

while

and Ada that allow passing of argument type information admit simpler declarations and fewer entry points.

void Normal3{fd}v(T arg);

means the two declarations

void Normal3fv(float arg[3]); void Normal3dv(double arg[3]);

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of 14 types (or pointers to one of these). These types are summarized in Table 2.2.

2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Most commands may be accumulated in a *display list* for processing by the GL at a later time. Otherwise, commands are effectively sent through a processing pipeline.

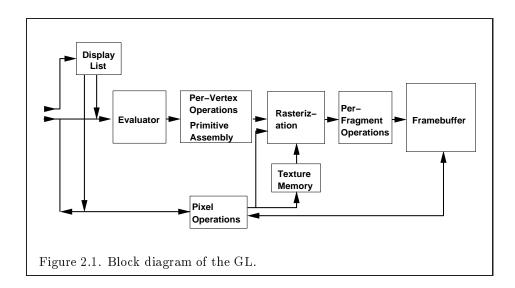
The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values. The next stage operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices are transformed and lit, and primitives are clipped to a viewing volume in preparation for the next stage, rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each *fragment* so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values.

Finally, there is a way to bypass the vertex processing portion of the pipeline to send a block of fragments directly to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer; values may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

This ordering is meant only as a tool for describing the GL, not as a strict rule of how the GL is implemented, and we present it only as a means to

GL Type	Minimum Number of Bits	Description		
boolean	1	Boolean		
byte	8	signed 2's complement binary inte-		
		ger		
ubyte	8	unsigned binary integer		
short	16	signed 2's complement binary inte-		
		ger		
ushort	16	unsigned binary integer		
int	32	signed 2's complement binary inte-		
		ger		
uint	32	unsigned binary integer		
sizei	32	Non-negative binary integer size		
enum	32	Enumerated binary integer value		
bitfield	32	Bit field		
float	32	Floating-point value		
clampf	32	Floating-point value clamped to		
		[0,1]		
double	64	Floating-point value		
clampd	64	Floating-point value clamped to		
		[0,1]		

Table 2.2: GL data types. GL types are not C types. Thus, for example, GL type int is referred to as GLint outside this document, and is not necessarily equivalent to the C type int. An implementation may use more bits than the number indicated in the table to represent a GL type. Correct interpretation of integer values outside the minimum range is not required, however.



organize the various operations of the GL. Objects such as curved surfaces, for instance, may be transformed before they are converted to polygons.

2.5 GL Errors

The GL detects only a subset of those conditions that could be considered errors. This is because in many cases error checking would adversely impact the performance of an error-free program.

The command

```
enum GetError( void );
```

is used to obtain error information. Each detectable error is assigned a numeric code. When an error is detected, a flag is set and the code is recorded. Further errors, if they occur, do not affect this recorded code. When **GetError** is called, the code is returned and the flag is cleared, so that a further error will again record its code. If a call to **GetError** returns NO_ERROR, then there has been no detectable error since the last call to **GetError** (or since the GL was initialized).

To allow for distributed implementations, there may be several flagcode pairs. In this case, after a call to **GetError** returns a value other than NO_ERROR each subsequent call returns the non-zero code of a distinct flag-code pair (in unspecified order), until all non-NO_ERROR codes have been returned. When there are no more non-NO_ERROR error codes, all flags are reset. This scheme requires some positive number of pairs of a flag bit and an integer. The initial state of all flags is cleared and the initial value of all codes is NO_ERROR.

Table 2.3 summarizes GL errors. Currently, when an error flag is set, results of GL operation are undefined only if OUT_OF_MEMORY has occurred. In other cases, the command generating the error is ignored so that it has no effect on GL state or framebuffer contents. If the generating command returns a value, it returns zero. If the generating command modifies values through a pointer argument, no change is made to these values. These error semantics apply only to GL errors, not to system errors such as memory access errors. This behavior is the current behavior; the action of the GL in the presence of errors is subject to change.

Three error generation conditions are implicit in the description of every GL command. First, if a command that requires an enumerated value is passed a symbolic constant that is not one of those specified as allowable for that command, the error INVALID_ENUM results. This is the case even if the argument is a pointer to a symbolic constant if that value is not allowable for the given command. Second, if a negative number is provided where an argument of type sizei is specified, the error INVALID_VALUE results. Finally, if memory is exhausted as a side effect of the execution of a command, the error OUT_OF_MEMORY may be generated. Otherwise errors are generated only for conditions that are explicitly described in this specification.

2.6 Begin/End Paradigm

In the GL, most geometric objects are drawn by enclosing a series of coordinate sets that specify vertices and optionally normals, texture coordinates, and colors between **Begin/End** pairs. There are ten geometric objects that are drawn this way: points, line segments, line segment loops, separated line segments, polygons, triangle strips, triangle fans, separated triangles, quadrilateral strips, and separated quadrilaterals.

Each vertex is specified with two, three, or four coordinates. In addition, a *current normal*, *current texture coordinates*, and *current color* may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive.

Primary and secondary colors are associated with each vertex (see sec-

Error	Description	Offending com-
		mand ignored?
INVALID_ENUM	enum argument out of range	Yes
INVALID_VALUE	Numeric argument out of	Yes
	range	
INVALID_OPERATION	Operation illegal in current	Yes
	state	
STACK_OVERFLOW	Command would cause a stack	Yes
	overflow	
STACK_UNDERFLOW	Command would cause a stack	Yes
	underflow	
OUT_OF_MEMORY	Not enough memory left to ex-	Unknown
	ecute command	
TABLE_TOO_LARGE	The specified table is too large	Yes

Table 2.3: Summary of GL errors

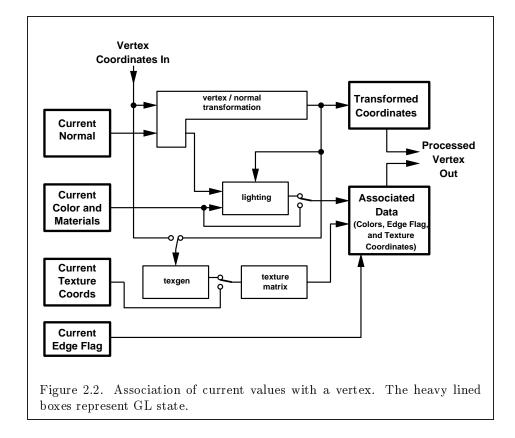
tion 3.9). These associated colors are either based on the current color or produced by lighting, depending on whether or not lighting is enabled. Texture coordinates are similarly associated with each vertex. Figure 2.2 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex*.

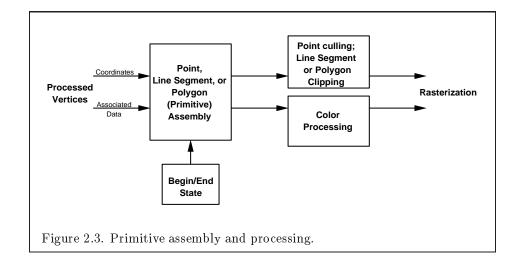
The current values are part of GL state. Vertices and normals are transformed, colors may be affected or replaced by lighting, and texture coordinates are transformed and possibly affected by a texture coordinate generation function. The processing indicated for each current value is applied for each vertex that is sent to the GL.

The methods by which vertices, normals, texture coordinates, and colors are sent to the GL, as well as how normals are transformed and how vertices are mapped to the two-dimensional screen, are discussed later.

Before colors have been assigned to a vertex, the state required by a vertex is the vertex's coordinates, the current normal, the current edge flag (see section 2.6.2), the current material properties (see section 2.13.2), and the current texture coordinates. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its edge flag, its assigned colors, and its texture coordinates.

Figure 2.3 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices. After a primi-





tive is formed, it is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates, texture coordinates, and colors. In the case of a polygon primitive, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have texture coordinates and colors associated with them.

2.6.1 Begin and End Objects

Begin and **End** require one state variable with eleven values: one value for each of the ten possible **Begin/End** objects, and one other value indicating that no **Begin/End** object is being processed. The two relevant commands are

void Begin(enum mode); void End(void);

There is no limit on the number of vertices that may be specified between a **Begin** and an **End**.

Points. A series of individual points may be specified by calling **Begin** with an argument value of **POINTS**. No special state need be kept between **Begin** and **End** in this case, since each point is independent of previous and following points.

Line Strips. A series of one or more connected line segments is specified by enclosing a series of two or more endpoints within a **Begin/End** pair when **Begin** is called with LINE_STRIP. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the *i*th vertex (for i > 1) specifies the beginning of the *i*th segment and the end of the i - 1st. The last vertex specifies the end of the last segment. If only one vertex is specified between the **Begin/End** pair, then no primitive is generated.

The required state consists of the processed vertex produced from the last vertex that was sent (so that a line segment can be generated from it to the current vertex), and a boolean flag indicating if the current vertex is the first vertex.

Line Loops. Line loops, specified with the LINE_LOOP argument value to **Begin**, are the same as line strips except that a final segment is added from the final specified vertex to the first vertex. The additional state consists of the processed first vertex.

Separate Lines. Individual line segments, each specified by a pair of vertices, are generated by surrounding vertex pairs with **Begin** and **End**

when the value of the argument to **Begin** is LINES. In this case, the first two vertices between a **Begin** and **End** pair define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of specified vertices is odd, then the last one is ignored. The state required is the same as for lines but it is used differently: a vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

Polygons. A polygon is described by specifying its boundary as a series of line segments. When **Begin** is called with **POLYGON**, the bounding line segments are specified in the same way as line loops. Depending on the current state of the GL, a polygon may be rendered in one of several ways such as outlining its border or filling its interior. A polygon described with fewer than three vertices does not generate a primitive.

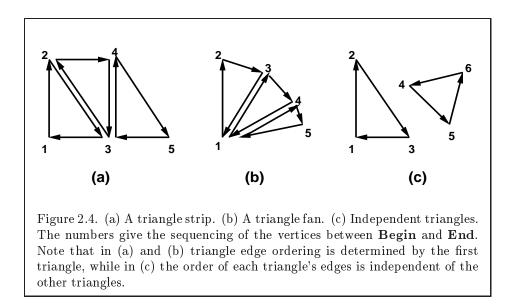
Only convex polygons are guaranteed to be drawn correctly by the GL. If a specified polygon is nonconvex when projected onto the window, then the rendered polygon need only lie within the convex hull of the projected vertices defining its boundary.

The state required to support polygons consists of at least two processed vertices (more than two are never required, although an implementation may use more); this is because a convex polygon can be rasterized as its vertices arrive, before all of them have been specified. The order of the vertices is significant in lighting and polygon rasterization (see sections 2.13.1 and 3.5.1).

Triangle strips. A triangle strip is a series of triangles connected along shared edges. A triangle strip is specified by giving a series of defining vertices between a **Begin/End** pair when **Begin** is called with TRIANGLE_STRIP. In this case, the first three vertices define the first triangle (and their order is significant, just as for polygons). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. A **Begin/End** pair enclosing fewer than three vertices, when TRIANGLE_STRIP has been supplied to **Begin**, produces no primitive. See Figure 2.4.

The state required to support triangle strips consists of a flag indicating if the first triangle has been completed, two stored processed vertices, (called vertex A and vertex B), and a one bit pointer indicating which stored vertex will be replaced with the next vertex. After a **Begin(TRIANGLE_STRIP)**, the pointer is initialized to point to vertex A. Each vertex sent between a **Begin/End** pair toggles the pointer. Therefore, the first vertex is stored as vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

Triangle fans. A triangle fan is the same as a triangle strip with one

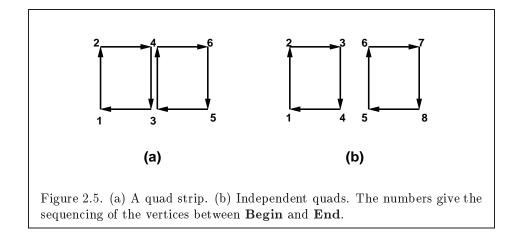


exception: each vertex after the first always replaces vertex B of the two stored vertices. The vertices of a triangle fan are enclosed between **Begin** and **End** when the value of the argument to **Begin** is **TRIANGLE_FAN**.

Separate Triangles. Separate triangles are specified by placing vertices between **Begin** and **End** when the value of the argument to **Begin** is TRIANGLES. In this case, The 3i + 1st, 3i + 2nd, and 3i + 3rd vertices (in that order) determine a triangle for each i = 0, 1, ..., n-1, where there are 3n + k vertices between the **Begin** and **End**. k is either 0, 1, or 2; if k is not zero, the final k vertices are ignored. For each triangle, vertex A is vertex 3i and vertex B is vertex 3i + 1. Otherwise, separate triangles are the same as a triangle strip.

The rules given for polygons also apply to each triangle generated from a triangle strip, triangle fan or from separate triangles.

Quadrilateral (quad) strips. Quad strips generate a series of edgesharing quadrilaterals from vertices appearing between **Begin** and **End**, when **Begin** is called with QUAD_STRIP. If the *m* vertices between the **Begin** and **End** are v_1, \ldots, v_m , where v_j is the *j*th specified vertex, then quad *i* has vertices (in order) v_{2i} , v_{2i+1} , v_{2i+3} , and v_{2i+2} with $i = 0, \ldots, \lfloor m/2 \rfloor$. The state required is thus three processed vertices, to store the last two vertices of the previous quad along with the third vertex (the first new vertex) of the current quad, a flag to indicate when the first quad has been completed, and a one-bit counter to count members of a vertex pair. See Figure 2.5.



A quad strip with fewer than four vertices generates no primitive. If the number of vertices specified for a quadrilateral strip between **Begin** and **End** is odd, the final vertex is ignored.

Separate Quadrilaterals Separate quads are just like quad strips except that each group of four vertices, the 4j + 1st, the 4j + 2nd, the 4j + 3rd, and the 4j + 4th, generate a single quad, for j = 0, 1, ..., n - 1. The total number of vertices between **Begin** and **End** is 4n + k, where $0 \le k \le 3$; if k is not zero, the final k vertices are ignored. Separate quads are generated by calling **Begin** with the argument value **QUADS**.

The rules given for polygons also apply to each quad generated in a quad strip or from separate quads.

2.6.2 Polygon Edges

Each edge of each primitive generated from a polygon, triangle strip, triangle fan, separate triangle set, quadrilateral strip, or separate quadrilateral set, is flagged as either *boundary* or *non-boundary*. These classifications are used during polygon rasterization; some modes affect the interpretation of polygon boundary edges (see section 3.5.4). By default, all edges are boundary edges, but the flagging of polygons, separate triangles, or separate quadrilaterals may be altered by calling

```
void EdgeFlag( boolean flag );
void EdgeFlagv( boolean *flag );
```

to change the value of a flag bit. If flag is zero, then the flag bit is set to FALSE; if flag is non-zero, then the flag bit is set to TRUE.

When **Begin** is supplied with one of the argument values POLYGON, TRIANGLES, or QUADS, each vertex specified within a **Begin** and **End** pair begins an edge. If the edge flag bit is TRUE, then each specified vertex begins an edge that is flagged as boundary. If the bit is FALSE, then induced edges are flagged as non-boundary.

The state required for edge flagging consists of one current flag bit. Initially, the bit is TRUE. In addition, each processed vertex of an assembled polygonal primitive must be augmented with a bit indicating whether or not the edge beginning on that vertex is boundary or non-boundary.

2.6.3 GL Commands within Begin/End

The only GL commands that are allowed within any **Begin/End** pairs are the commands for specifying vertex coordinates, vertex color, normal coordinates, and texture coordinates (Vertex, Color, Index, Normal, Tex-Coord), the ArrayElement command (see section 2.8), the EvalCoord and EvalPoint commands (see section 5.1), commands for specifying lighting material parameters (Material commands; see section 2.13.2), display list invocation commands (CallList and CallLists; see section 5.4), and the EdgeFlag command. Executing any other GL command between the execution of Begin and the corresponding execution of End results in the error INVALID_OPERATION. Executing Begin after Begin has already been executed but before an End is executed generates the INVALID_OPERATION error, as does executing End without a previous corresponding Begin.

Execution of the commands EnableClientState, DisableClientState, PushClientAttrib, PopClientAttrib, EdgeFlag-Pointer, TexCoordPointer, ColorPointer, IndexPointer, Normal-Pointer, VertexPointer, InterleavedArrays, and PixelStore, is not allowed within any Begin/End pair, but an error may or may not be generated if such execution occurs. If an error is not generated, GL operation is undefined. (These commands are described in sections 2.8, 3.6.1, and Chapter 6.)

2.7 Vertex Specification

Vertices are specified by giving their coordinates in two, three, or four dimensions. This is done using one of several versions of the **Vertex** command:

```
void Vertex{234}{sifd}( T coords );
void Vertex{234}{sifd}v( T coords );
```

A call to any **Vertex** command specifies four coordinates: x, y, z, and w. The x coordinate is the first coordinate, y is second, z is third, and w is fourth. A call to **Vertex2** sets the x and y coordinates; the z coordinate is implicitly set to zero and the w coordinate to one. **Vertex3** sets x, y, and z to the provided values and w to one. **Vertex4** sets all four coordinates, allowing the specification of an arbitrary point in projective three-space. Invoking a **Vertex** command outside of a **Begin/End** pair results in undefined behavior.

Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

```
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( T coords );
```

specify the current homogeneous texture coordinates, named s, t, r, and q. The **TexCoord1** family of commands set the s coordinate to the provided single argument while setting t and r to 0 and q to 1. Similarly, **TexCoord2** sets s and t to the specified values, r to 0 and q to 1; **TexCoord3** sets s, t, and r, with q set to 1, and **TexCoord4** sets all four texture coordinates.

The current normal is set using

```
void Normal3{bsifd}( T coords );
void Normal3{bsifd}v( T coords );
```

Byte, short, or integer values passed to **Normal** are converted to floatingpoint values as indicated for the corresponding (signed) type in Table 2.6.

Finally, there are several ways to set the current color. The GL stores both a current single-valued *color index*, and a current four-valued RGBA color. One or the other of these is significant depending as the GL is in *color index mode* or *RGBA mode*. The mode selection is made when the GL is initialized.

The command to set RGBA colors is

```
void Color{34}{bsifd ubusui}( T components );
void Color{34}{bsifd ubusui}v( T components );
```

The **Color** command has two major variants: **Color3** and **Color4**. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.13.)

Versions of the **Color** command that take floating-point values accept values nominally between 0.0 and 1.0. 0.0 corresponds to the minimum while 1.0 corresponds to the maximum (machine dependent) value that a component may take on in the framebuffer (see section 2.13 on colors and coloring). Values outside [0, 1] are not clamped.

The command

```
void Index{sifd ub}( T index );
void Index{sifd ub}v( T index );
```

updates the current (single-valued) color index. It takes one argument, the value to which the current color index should be set. Values outside the (machine-dependent) representable range of color indices are not clamped.

The state required to support vertex specification consists of four floating-point numbers to store the current texture coordinates s, t, r, and q, three floating-point numbers to store the three coordinates of the current normal, four floating-point values to store the current RGBA color, and one floating-point value to store the current color index. There is no notion of a current vertex, so no state is devoted to vertex coordinates. The initial values of s, t, and r of the current texture coordinates are zero; the initial value of q is one. The initial current normal has coordinates (0, 0, 1). The initial RGBA color is $(\mathbf{R}, \mathbf{G}, \mathbf{B}, \mathbf{A}) = (1, 1, 1, 1)$. The initial color index is 1.

2.8 Vertex Arrays

The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to six arrays: one each to store edge flags, texture coordinates, colors, color indices, normals, and vertices. The commands

- void EdgeFlagPointer(sizei stride, void *pointer);
- void TexCoordPointer(int size, enum type, sizei stride, void *pointer);
- void ColorPointer(int size, enum type, sizei stride, void *pointer);

Command	Sizes	Types
VertexPointer	2,3,4	short, int, float, double
NormalPointer	3	byte, short, int, float, double
ColorPointer	3,4	byte, ubyte, short, ushort, int,
		uint, float, double
IndexPointer	1	ubyte, short, int, float, double
TexCoordPointer	1,2,3,4	<pre>short, int, float, double</pre>
EdgeFlagPointer	1	boolean

Table 2.4: Vertex array sizes (values per vertex) and data types.

void IndexPointer(enum type, sizei stride, void *pointer);

void NormalPointer(enum type, sizei stride, void *pointer);

void VertexPointer(int size, enum type, sizei stride, void *pointer);

describe the locations and organizations of these arrays. For each command, type specifies the data type of the values stored in the array. Because edge flags are always type boolean, EdgeFlagPointer has no type argument. size, when present, indicates the number of values per vertex that are stored in the array. Because normals are always specified with three values, NormalPointer has no size argument. Likewise, because color indices and edge flags are always specified with a single value, IndexPointer and EdgeFlagPointer also have no size argument. Table 2.4 indicates the allowable values for size and type (when present). For type the values BYTE, SHORT, INT, FLOAT, and DOUBLE indicate types byte, short, int, float, and double, respectively; and the values UNSIGNED_BYTE, UNSIGNED_SHORT, and UNSIGNED_INT indicate types ubyte, ushort, and uint, respectively. The error INVALID_VALUE is generated if size is specified with a value other than that indicated in the table.

The one, two, three, or four values in an array that correspond to a single vertex comprise an array *element*. The values within each array element are stored sequentially in memory. If *stride* is specified as zero, then array elements are stored sequentially as well. Otherwise pointers to the *i*th and (i + 1)st elements of an array differ by *stride* basic machine units (typically

unsigned bytes), the pointer to the (i + 1)st element being greater. For each command, *pointer* specifies the location in memory of the first value of the first element of the array being specified.

An individual array is enabled or disabled by calling one of

void EnableClientState(enum array); void DisableClientState(enum array);

with *array* set to EDGE_FLAG_ARRAY, TEXTURE_COORD_ARRAY, COLOR_ARRAY, INDEX_ARRAY, NORMAL_ARRAY, or VERTEX_ARRAY, for the edge flag, texture coordinate, color, color index, normal, or vertex array, respectively.

The *i*th element of every enabled array is transferred to the GL by calling

```
void ArrayElement( int i );
```

For each enabled array, it is as though the corresponding command from section 2.7 or section 2.6.2 were called with a pointer to element *i*. For the vertex array, the corresponding command is Vertex[size][type]v, where size is one of [2,3,4], and type is one of [s,i,f,d], corresponding to array types short, int, float, and double respectively. The corresponding commands for the edge flag, texture coordinate, color, color index, and normal arrays are EdgeFlagv, TexCoord[size][type]v, Color[size][type]v, Index[type]v, and Normal[type]v, respectively. If the vertex array is enabled, it is as though Vertex[size][type]v is executed last, after the executions of the other corresponding commands.

Changes made to array data between the execution of **Begin** and the corresponding execution of **End** may affect calls to **ArrayElement** that are made within the same **Begin/End** period in non-sequential ways. That is, a call to **ArrayElement** that precedes a change to array data may access the changed data, and a call that follows a change to array data may access original data.

The command

void DrawArrays(enum mode, int first, sizei count);

constructs a sequence of geometric primitives using elements first through first+count-1 of each enabled array. mode specifies what kind of primitives are constructed; it accepts the same token values as the mode parameter of the **Begin** command. The effect of

DrawArrays (mode, first, count);

is the same as the effect of the command sequence

```
if (mode or count is invalid )
  generate appropriate error
else {
    int i;
    Begin(mode);
    for (i=0; i < count; i++)
        ArrayElement(first+i);
    End();
}</pre>
```

with one exception: the current edge flag, texture coordinates, color, color index, and normal coordinates are each indeterminate after the execution of **DrawArrays**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawArrays**.

The command

```
void DrawElements( enum mode, sizei count, enum type,
void *indices );
```

constructs a sequence of geometric primitives using the *count* elements whose indices are stored in *indices*. *type* must be one of UNSIGNED_BYTE, UNSIGNED_SHORT, or UNSIGNED_INT, indicating that the values in *indices* are indices of GL type ubyte, ushort, or uint respectively. *mode* specifies what kind of primitives are constructed; it accepts the same token values as the *mode* parameter of the **Begin** command. The effect of

DrawElements (mode, count, type, indices);

is the same as the effect of the command sequence

```
if (mode, count, or type is invalid )
  generate appropriate error
else {
    int i;
    Begin(mode);
    for (i=0; i < count; i++)
        ArrayElement(indices[i]);
    End();
}</pre>
```

with one exception: the current edge flag, texture coordinates, color, color index, and normal coordinates are each indeterminate after the execution of **DrawElements**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawElements**.

The command

```
void DrawRangeElements( enum mode, uint start,
    uint end, sizei count, enum type, void *indices );
```

is a restricted form of **DrawElements**. mode, count, type, and indices match the corresponding arguments to **DrawElements**, with the additional constraint that all values in the array indices must lie between start and end inclusive.

Implementations denote recommended maximum amounts of vertex and index data, which may be queried by calling **GetIntegerv** with the symbolic constants MAX_ELEMENTS_VERTICES and MAX_ELEMENTS_INDICES. If end-start+1is greater than the value of MAX_ELEMENTS_VERTICES, or if *count* is greater than the value of MAX_ELEMENTS_INDICES, then the call may operate at reduced performance. There is no requirement that all vertices in the range [*start*, *end*] be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.

The error INVALID_VALUE is generated if end < start. Invalid mode, count, or type parameters generate the same errors as would the corresponding call to **DrawElements**. It is an error for indices to lie outside the range [start, end], but implementations may not check for this. Such indices will cause implementation-dependent behavior.

The command

void InterleavedArrays(enum format, sizei stride, void *pointer);

efficiently initializes the six arrays and their enables to one of 14 configurations. *format* must be one of 14 symbolic constants: V2F, V3F, C4UB_V2F, C4UB_V3F, C3F_V3F, N3F_V3F, C4F_N3F_V3F, T2F_V3F, T4F_V4F, T2F_C4UB_V3F, T2F_C3F_V3F, T2F_N3F_V3F, T2F_C4F_N3F_V3F, or T4F_C4F_N3F_V4F.

The effect of

InterleavedArrays(*format*, *stride*, *pointer*);

is the same as the effect of the command sequence

format	e_t	e_c	e_n	s_t	s_c	s_v	t_c
V2F	False	False	False			2	
V3F	False	False	False			3	
C4UB_V2F	False	True	False		4	2	UNSIGNED_BYTE
C4UB_V3F	False	True	False		4	3	UNSIGNED_BYTE
C3F_V3F	False	True	False		3	3	FLOAT
N3F_V3F	False	False	True			3	
C4F_N3F_V3F	False	True	True		4	3	FLOAT
T2F_V3F	True	False	False	2		3	
T4F_V4F	True	False	False	4		4	
T2F_C4UB_V3F	True	True	False	2	4	3	UNSIGNED_BYTE
T2F_C3F_V3F	True	True	False	2	3	3	FLOAT
T2F_N3F_V3F	True	False	True	2		3	
T2F_C4F_N3F_V3F	True	True	True	2	4	3	FLOAT
T4F_C4F_N3F_V4F	True	True	True	4	4	4	FLOAT

format	p_c	p_n	p_v	\$
V2F			0	2f
V3F			0	3f
C4UB_V2F	0		С	c+2f
C4UB_V3F	0		С	c + 3f
C3F_V3F	0		3f	6f
N3F_V3F		0	3f	6f
C4F_N3F_V3F	0	4f	7f	10f
T2F_V3F			2f	5f
T4F_V4F			4f	8f
T2F_C4UB_V3F	2f		c+2f	c+5f
T2F_C3F_V3F	2f		5f	8f
T2F_N3F_V3F		2f	5f	8f
T2F_C4F_N3F_V3F	2f	6f	9f	12f
T4F_C4F_N3F_V4F	4f	8f	11f	15f

Table 2.5: Variables that direct the execution of InterleavedArrays. f is sizeof(FLOAT). c is 4 times sizeof(UNSIGNED_BYTE), rounded up to the nearest multiple of f. All pointer arithmetic is performed in units of sizeof(UNSIGNED_BYTE).

```
if (format or stride is invalid)
   generate appropriate error
else {
   int str;
   set e_t, e_c, e_n, s_t, s_c, s_v, t_c, p_c, p_n, p_v, and s as a function
      of Table 2.5 and the value of format.
   str = stride;
   if (str is zero)
      str = s;
   DisableClientState(EDGE_FLAG_ARRAY);
   DisableClientState(INDEX_ARRAY);
   if (e_t) {
      EnableClientState(TEXTURE_COORD_ARRAY);
      TexCoordPointer(s_t, FLOAT, str, pointer);
   } else {
      DisableClientState(TEXTURE_COORD_ARRAY);
   }
   if (e_c) {
      EnableClientState(COLOR_ARRAY);
      ColorPointer (s_c, t_c, \text{str}, pointer + p_c);
   } else {
      DisableClientState(COLOR_ARRAY);
   }
   if (e_n) {
      EnableClientState(NORMAL_ARRAY);
      NormalPointer (FLOAT, str, pointer + p_n);
   } else {
      DisableClientState(NORMAL_ARRAY);
   }
   EnableClientState(VERTEX_ARRAY);
   VertexPointer (s_v, \text{FLOAT}, \text{str}, pointer + p_v);
}
```

The client state required to implement vertex arrays consists of six boolean values, six memory pointers, six integer stride values, five symbolic constants representing array types, and three integers representing values per element. In the initial state the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four.

2.9 Rectangles

There is a set of GL commands to support efficient specification of rectangles as two corner vertices.

```
void Rect{sifd}( T x1, T y1, T x2, T y2);
void Rect{sifd}v( T v1[2], T v2[2]);
```

Each command takes either four arguments organized as two consecutive pairs of (x, y) coordinates, or two pointers to arrays each of which contains an x value followed by a y value. The effect of the **Rect** command

Rect (x_1, y_1, x_2, y_2) ;

is exactly the same as the following sequence of commands:

```
Begin (POLYGON);

Vertex2(x_1, y_1);

Vertex2(x_2, y_1);

Vertex2(x_2, y_2);

Vertex2(x_1, y_2);

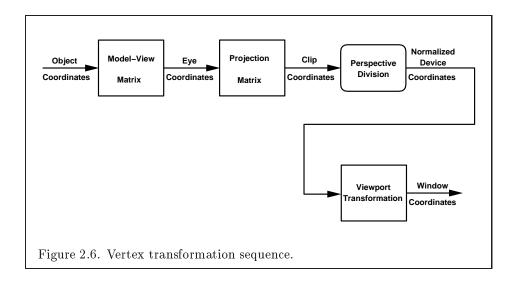
End();
```

The appropriate **Vertex2** command would be invoked depending on which of the **Rect** commands is issued.

2.10 Coordinate Transformations

Vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how this transformation is controlled.

Figure 2.6 diagrams the sequence of transformations that are applied to vertices. The vertex coordinates that are presented to the GL are termed *object coordinates*. The *model-view* matrix is applied to these coordinates to yield *eye* coordinates. Then another matrix, called the *projection* matrix, is applied to eye coordinates to yield *clip* coordinates. A perspective division is carried out on clip coordinates to yield *normalized device* coordinates. A final *viewport* transformation is applied to convert these coordinates into *window coordinates*.



Object coordinates, eye coordinates, and clip coordinates are fourdimensional, consisting of x, y, z, and w coordinates (in that order). The model-view and perspective matrices are thus 4×4 .

If a vertex in object coordinates is given by $\begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$ and the model-view

matrix is M, then the vertex's eye coordinates are found as

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = M \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}.$$

Similarly, if ${\cal P}$ is the projection matrix, then the vertex's clip coordinates are

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}.$$

The vertex's normalized device coordinates are then

$$egin{pmatrix} x_d \ y_d \ z_d \end{pmatrix} = egin{pmatrix} x_c/w_c \ y_c/w_c \ z_c/w_c \end{pmatrix}.$$

2.10.1 Controlling the Viewport

The viewport transformation is determined by the viewport's width and height in pixels, p_x and p_y , respectively, and its center (o_x, o_y) (also in

pixels). The vertex's window coordinates, $\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix}$, are given by

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} (p_x/2)x_d + o_x \\ (p_y/2)y_d + o_y \\ [(f-n)/2]z_d + (n+f)/2 \end{pmatrix}.$$

The factor and offset applied to z_d encoded by n and f are set using

```
void DepthRange( clampd n, clampd f );
```

Each of n and f are clamped to lie within [0, 1], as are all arguments of type clampd or clampf. z_w is taken to be represented in fixed-point with at least as many bits as there are in the depth buffer of the framebuffer. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

Viewport transformation parameters are specified using

void Viewport(int x, int y, sizei w, sizei h);

where x and y give the x and y window coordinates of the viewport's lowerleft corner and w and h give the viewport's width and height, respectively. The viewport parameters shown in the above equations are found from these values as $o_x = x + w/2$ and $o_y = y + h/2$; $p_x = w$, $p_y = h$.

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by issuing an appropriate **Get** command (see Chapter 6). The maximum viewport dimensions must be greater than or equal to the visible dimensions of the display being rendered to. INVALID_VALUE is generated if either w or h is negative.

The state required to implement the viewport transformation is 6 integers. In the initial state, w and h are set to the width and height, respectively, of the window into which the GL is to do its rendering. o_x and o_y are set to w/2 and h/2, respectively. n and f are set to 0.0 and 1.0, respectively.

2.10.2 Matrices

The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

```
void MatrixMode( enum mode );
```

which takes one of the pre-defined constants TEXTURE, MODELVIEW, COLOR, or PROJECTION as the argument value. TEXTURE is described later in section 2.10.2, and COLORis described in section 3.6.3. If the current matrix mode is MODELVIEW, then matrix operations apply to the model-view matrix; if PROJECTION, then they apply to the projection matrix.

The two basic commands for affecting the current matrix are

void LoadMatrix{fd}(T m[16]); void MultMatrix{fd}(T m[16]);

LoadMatrix takes a pointer to a 4×4 matrix stored in column-major order as 16 consecutive floating-point values, i.e. as

$$egin{pmatrix} a_1 & a_5 & a_9 & a_{13} \ a_2 & a_6 & a_{10} & a_{14} \ a_3 & a_7 & a_{11} & a_{15} \ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}.$$

(This differs from the standard row-major C ordering for matrix elements. If the standard ordering is used, all of the subsequent transformation equations are transposed, and the columns representing vectors become rows.)

The specified matrix replaces the current matrix with the one pointed to. **MultMatrix** takes the same type argument as **LoadMatrix**, but multiplies the current matrix by the one pointed to and replaces the current matrix with the product. If C is the current matrix and M is the matrix pointed to by **MultMatrix**'s argument, then the resulting current matrix, C', is

$$C' = C \cdot M.$$

The command

void LoadIdentity(void);

effectively calls LoadMatrix with the identity matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

There are a variety of other commands that manipulate matrices. **Ro**tate, **Translate**, **Scale**, **Frustum**, and **Ortho** manipulate the current matrix. Each computes a matrix and then invokes **MultMatrix** with this matrix. In the case of

void Rotate{fd}(T
$$\theta$$
, T x , T y , T z);

 θ gives an angle of rotation in degrees; the coordinates of a vector **v** are given by $\mathbf{v} = (x \ y \ z)^T$. The computed matrix is a counter-clockwise rotation about the line through the origin with the specified axis when that axis is pointing up (i.e. the right-hand rule determines the sense of the rotation angle). The matrix is thus

$$egin{pmatrix} & & & 0 \ & R & & 0 \ & & & 0 \ 0 & 0 & 0 & 1 \end{pmatrix}$$

Let $\mathbf{u} = \mathbf{v}/||\mathbf{v}|| = (x' \ y' \ z')^T$. If

$$S = egin{pmatrix} 0 & -z' & y' \ z' & 0 & -x' \ -y' & x' & 0 \end{pmatrix}$$

then

$$R = \mathbf{u}\mathbf{u}^T + \cos\theta(I - \mathbf{u}\mathbf{u}^T) + \sin\theta S$$

The arguments to

void **Translate**{ \mathbf{fd} }(T x, T y, T z);

give the coordinates of a translation vector as $(x \ y \ z)^T$. The resulting matrix is a translation by the specified vector:

/1	0	0	$x \setminus$
0	1	0	y
0	0	1	z ·
$\setminus 0$	0	0	1/

void Scale{fd}(T x, T y, T z);

produces a general scaling along the x-, y-, and z- axes. The corresponding matrix is (x = 0, 0, 0)

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For

the coordinates $(l \ b \ -n)^T$ and $(r \ t \ -n)^T$ specify the points on the near clipping plane that are mapped to the lower-left and upper-right corners of the window, respectively (assuming that the eye is located at $(0 \ 0 \ 0)^T$). f gives the distance from the eye to the far clipping plane. If either n or f is less than or equal to zero, l is equal to r, b is equal to t, or n is equal to f, the error INVALID_VALUE results. The corresponding matrix is

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0\\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0\\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n}\\ 0 & 0 & -1 & 0 \end{pmatrix}$$

describes a matrix that produces parallel projection. $(l \ b - n)^T$ and $(r \ t - n)^T$ specify the points on the near clipping plane that are mapped to the lowerleft and upper-right corners of the window, respectively. f gives the distance from the eye to the far clipping plane. If l is equal to r, b is equal to t, or nis equal to f, the error INVALID_VALUE results. The corresponding matrix is

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

There is another 4×4 matrix that is applied to texture coordinates. This matrix is applied as

$$egin{pmatrix} m_1 & m_5 & m_9 & m_{13} \ m_2 & m_6 & m_{10} & m_{14} \ m_3 & m_7 & m_{11} & m_{15} \ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} egin{pmatrix} s \ t \ r \ q \end{pmatrix},$$

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to **TEXTURE** causes the already described matrix operations to apply to the texture matrix.

There is a stack of matrices for each of the matrix modes. For MODELVIEW mode, the stack depth is at least 32 (that is, there is a stack of at least 32 model-view matrices). For the other modes, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

```
void PushMatrix( void );
```

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

void PopMatrix(void);

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error STACK_UNDERFLOW; pushing a matrix onto a full stack generates STACK_OVERFLOW.

The state required to implement transformations consists of a fourvalued integer indicating the current matrix mode, a stack of at least two 4×4 matrices for each of COLOR, PROJECTION, and TEXTURE with associated stack pointers, and a stack of at least $32 \ 4 \times 4$ matrices with an associated stack pointer for MODELVIEW. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW.

2.10.3 Normal Transformation

Finally, we consider how the model-view matrix and transformation state affect normals. Before use in lighting, normals are transformed to eye coordinates by a matrix derived from the model-view matrix. Rescaling and normalization operations are performed on the transformed normals to make them unit length prior to use in lighting. Rescaling and normalization are controlled by

 and

with *target* equal to **RESCALE_NORMAL** or **NORMALIZE**. This requires two bits of state. The initial state is for normals not to be rescaled or normalized.

If the model-view matrix is M, then the normal is transformed to eye coordinates by:

$$(n_{x}' \quad n_{y}' \quad n_{z}' \quad q') = (n_{x} \quad n_{y} \quad n_{z} \quad q) \cdot M^{-1}$$
where, if $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$ are the associated vertex coordinates, then
$$q = \begin{cases} 0, & w = 0, \\ -(n_{x} \quad n_{y} \quad n_{z}) \begin{pmatrix} x \\ y \\ z \end{pmatrix}, & w \neq 0 \end{cases}$$
(2.1)

Implementations may choose instead to transform $\begin{pmatrix} n_x & n_y & n_z \end{pmatrix}$ to eye coordinates using

$$(n_x' \quad n_y' \quad n_z') = (n_x \quad n_y \quad n_z) \cdot M_u^{-1}$$

where M_u is the upper leftmost 3x3 matrix taken from M.

Rescale multiplies the transformed normals by a scale factor

$$(n_x'' \quad n_y'' \quad n_z'') = f(n_x' \quad n_y' \quad n_z')$$

If rescaling is disabled, then f = 1. If rescaling is enabled, then f is computed as $(m_{ij}$ denotes the matrix element in row i and column j of M^{-1} , numbering the topmost row of the matrix as row 1 and the leftmost column as column 1)

$$f = \frac{1}{\sqrt{m_{31}^2 + m_{32}^2 + m_{33}^2}}$$

Note that if the normals sent to GL were unit length and the model-view matrix uniformly scales space, then rescale makes the transformed normals unit length.

Alternatively, an implementation may chose f as

$$f = \frac{1}{\sqrt{{n_x}'^2 + {n_y}'^2 + {n_z}'^2}}$$

recomputing f for each normal. This makes all non-zero length normals unit length regardless of their input length and the nature of the modelview matrix.

After rescaling, the final transformed normal used in lighting, n_f , is computed as

$$n_f = m (n_x'' \quad n_y'' \quad n_z'')$$

If normalization is disabled, then m = 1. Otherwise

$$m = \frac{1}{\sqrt{n_x''^2 + n_y''^2 + n_z''^2}}$$

Because we specify neither the floating-point format nor the means for matrix inversion, we cannot specify behavior in the case of a poorlyconditioned (nearly singular) model-view matrix M. In case of an exactly singular matrix, the transformed normal is undefined. If the GL implementation determines that the model-view matrix is uninvertible, then the entries in the inverted matrix are arbitrary. In any case, neither normal transformation nor use of the transformed normal may lead to GL interruption or termination.

2.10.4 Generating Texture Coordinates

Texture coordinates associated with a vertex may either be taken from the current texture coordinates or generated according to a function dependent on vertex coordinates. The command

void TexGen{ifd}(enum coord, enum pname, T param); void TexGen{ifd}v(enum coord, enum pname, T params);

controls texture coordinate generation. *coord* must be one of the constants S, T, R, or Q, indicating that the pertinent coordinate is the s, t, r, or q

coordinate, respectively. In the first form of the command, *param* is a symbolic constant specifying a single-valued texture generation parameter; in the second form, *params* is a pointer to an array of values that specify texture generation parameters. *pname* must be one of the three symbolic constants TEXTURE_GEN_MODE, OBJECT_PLANE, or EYE_PLANE. If *pname* is TEXTURE_GEN_MODE, then either *params* points to or *param* is an integer that is one of the symbolic constants OBJECT_LINEAR, EYE_LINEAR, or SPHERE_MAP.

If TEXTURE_GEN_MODE indicates OBJECT_LINEAR, then the generation function for the coordinate indicated by *coord* is

$$g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o.$$

 x_o, y_o, z_o , and w_o are the object coordinates of the vertex. p_1, \ldots, p_4 are specified by calling **TexGen** with *pname* set to **OBJECT_PLANE** in which case *params* points to an array containing p_1, \ldots, p_4 . There is a distinct group of plane equation coefficients for each texture coordinate; *coord* indicates the coordinate to which the specified coefficients pertain.

If TEXTURE_GEN_MODE indicates EYE_LINEAR, then the function is

$$g = p_1' x_e + p_2' y_e + p_3' z_e + p_4' w_e$$

where

$$(p'_1 \quad p'_2 \quad p'_3 \quad p'_4) = (p_1 \quad p_2 \quad p_3 \quad p_4) M^{-1}$$

 x_e, y_e, z_e , and w_e are the eye coordinates of the vertex. p_1, \ldots, p_4 are set by calling **TexGen** with *pname* set to **EYE_PLANE** in correspondence with setting the coefficients in the **OBJECT_PLANE** case. M is the model-view matrix in effect when p_1, \ldots, p_4 are specified. Computed texture coordinates may be inaccurate or undefined if M is poorly conditioned or singular.

When used with a suitably constructed texture image, calling **TexGen** with **TEXTURE_GEN_MODE** indicating SPHERE_MAP can simulate the reflected image of a spherical environment on a polygon. SPHERE_MAP texture coordinates are generated as follows. Denote the unit vector pointing from the origin to the vertex (in eye coordinates) by **u**. Denote the current normal, after transformation to eye coordinates, by **n**'. Let $\mathbf{r} = (r_x \ r_y \ r_z)^T$, the reflection vector, be given by

$$\mathbf{r} = \mathbf{u} - 2\mathbf{n}^{\prime T} \left(\mathbf{n}^{\prime} \mathbf{u} \right),$$

and let $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$. Then the value assigned to an *s* coordinate (the first **TexGen** argument value is **S**) is $s = r_x/m + \frac{1}{2}$; the value

assigned to a t coordinate is $t = r_y/m + \frac{1}{2}$. Calling **TexGen** with a coord of either **R** or **Q** when pname indicates SPHERE_MAP generates the error INVALID_ENUM.

A texture coordinate generation function is enabled or disabled using **Enable** and **Disable** with an argument of **TEXTURE_GEN_S**, **TEXTURE_GEN_T**, **TEXTURE_GEN_R**, or **TEXTURE_GEN_Q** (each indicates the corresponding texture coordinate). When enabled, the specified texture coordinate is computed according to the current EYE_LINEAR, OBJECT_LINEAR or SPHERE_MAP specification, depending on the current setting of **TEXTURE_GEN_MODE** for that coordinate. When disabled, subsequent vertices will take the indicated texture coordinate from the current texture coordinates.

The state required for texture coordinate generation comprises a threevalued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR. The initial state has the texture generation function disabled for all texture coordinates. The initial values of p_i for s are all 0 except p_1 which is one; for t all the p_i are zero except p_2 , which is 1. The values of p_i for r and q are all 0. These values of p_i apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

2.11 Clipping

Primitives are clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by

$$egin{aligned} -w_c &\leq x_c \leq w_c \ -w_c &\leq y_c \leq w_c \ -w_c &\leq z_c \leq w_c \end{aligned}$$

This view volume may be further restricted by as many as n client-defined clip planes to generate the clip volume. (n is an implementation dependent maximum that must be at least 6.) Each client-defined plane specifies a half-space. The clip volume is the intersection of all such half-spaces with the view volume (if there no client-defined clip planes are enabled, the clip volume is the view volume).

A client-defined clip plane is specified with

void ClipPlane(enum p, double eqn[4]);

The value of the first argument, p, is a symbolic constant, CLIP_PLANE*i*, where i is an integer between 0 and n - 1, indicating one of n client-defined clip planes. eqn is an array of four double-precision floating-point values. These are the coefficients of a plane equation in object coordinates: p_1 , p_2 , p_3 , and p_4 (in that order). The inverse of the current model-view matrix is applied to these coefficients, at the time they are specified, yielding

$$(p'_1 \quad p'_2 \quad p'_3 \quad p'_4) = (p_1 \quad p_2 \quad p_3 \quad p_4) M^{-1}$$

(where M is the current model-view matrix; the resulting plane equation is undefined if M is singular and may be inaccurate if M is poorly-conditioned) to obtain the plane equation coefficients in eye coordinates. All points with eye coordinates $(x_e \ y_e \ z_e \ w_e)^T$ that satisfy

$$(\begin{array}{ccc} p_1' & p_2' & p_3' & p_4' \end{array}) \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

lie in the half-space defined by the plane; points that do not satisfy this condition do not lie in the half-space.

Client-defined clip planes are enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is CLIP_PLANE*i* where *i* is an integer between 0 and *n*; specifying a value of *i* enables or disables the plane equation with index *i*. The constants obey CLIP_PLANE $i = \text{CLIP}_{PLANE0} + i$.

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded. If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume and discards it if it lies entirely outside the volume. If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \le t \le 1$, for each clipped vertex. If the coordinates of a clipped vertex are **P** and the original vertices' coordinates are **P**₁ and **P**₂, then t is given by

$$\mathbf{P} = t\mathbf{P}_1 + (1-t)\mathbf{P}_2.$$

The value of t is used in color and texture coordinate clipping (section 2.13.8).

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a polygon. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag TRUE), and so that original edges of the polygon that become cut off at these vertices retain their original flags.

If it happens that a polygon intersects an edge of the clip volume's boundary, then the clipped polygon must include a point on this boundary edge. This point must lie in the intersection of the boundary edge and the convex hull of the vertices of the original polygon. We impose this requirement because the polygon may not be exactly planar.

A line segment or polygon whose vertices have w_c values of differing signs may generate multiple connected components after clipping. GL implementations are not required to handle this situation. That is, only the portion of the primitive that lies in the region of $w_c > 0$ need be produced by clipping.

Primitives rendered with clip planes must satisfy a complementarity criterion. Suppose a single clip plane with coefficients $(p'_1 \quad p'_2 \quad p'_3 \quad p'_4)$ (or a number of similarly specified clip planes) is enabled and a series of primitives are drawn. Next, suppose that the original clip plane is respecified with coefficients $(-p'_1 \quad -p'_2 \quad -p'_3 \quad -p'_4)$ (and correspondingly for any other clip planes) and the primitives are drawn again (and the GL is otherwise in the same state). In this case, primitives must not be missing any pixels, nor may any pixels be drawn twice in regions where those primitives are cut by the clip planes.

The state required for clipping is at least 6 sets of plane equations (each consisting of four double-precision floating-point coefficients) and at least 6 corresponding bits indicating which of these client-defined plane equations are enabled. In the initial state, all client-defined plane equation coefficients are zero and all planes are disabled.

2.12 Current Raster Position

The *current raster position* is used by commands that directly affect pixels in the framebuffer. These commands, which bypass vertex transformation and primitive assembly, are described in the next chapter. The current raster position, however, shares some of the characteristics of a vertex. The state required for the current raster position consists of three window coordinates x_w , y_w , and z_w , a clip coordinate w_c value, an eye coordinate distance, a valid bit, and associated data consisting of a color and texture coordinates. It is set using one of the **RasterPos** commands:

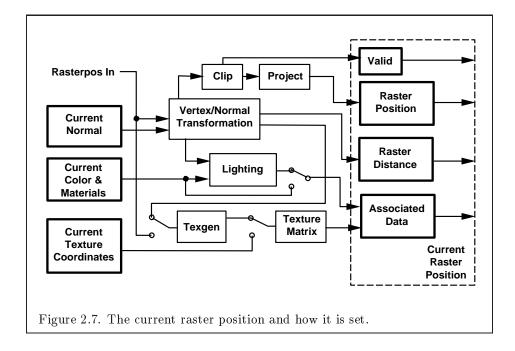
void RasterPos{234}{sifd}(T coords); void RasterPos{234}{sifd}v(T coords);

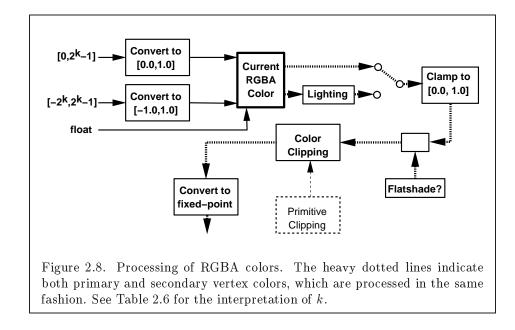
RasterPos4 takes four values indicating x, y, z, and w. **RasterPos3** (or **RasterPos2**) is analogous, but sets only x, y, and z with w implicitly set to 1 (or only x and y with z implicitly set to 0 and w implicitly set to 1).

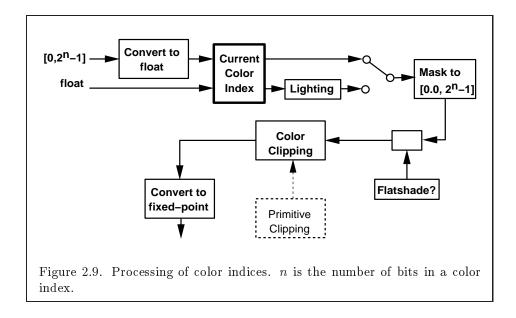
The coordinates are treated as if they were specified in a **Vertex** command. The x, y, z, and w coordinates are transformed by the current model-view and perspective matrices. These coordinates, along with current values, are used to generate a color and texture coordinates just as is done for a vertex. The color and texture coordinates so produced replace the color and texture coordinates stored in the current raster position's associated data. The distance from the origin of the eye coordinate system to the vertex as transformed by only the current model-view matrix replaces the current raster distance. This distance can be approximated (see section 3.10).

The transformed coordinates are passed to clipping as if they represented a point. If the "point" is not culled, then the projection to window coordinates is computed (section 2.10) and saved as the current raster position, and the valid bit is set. If the "point" is culled, the current raster position and its associated data become indeterminate and the valid bit is cleared. Figure 2.7 summarizes the behavior of the current raster position.

The current raster position requires five single-precision floating-point values for its x_w , y_w , and z_w window coordinates, its w_c clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA and color index), and texture coordinates for associated data. In the initial state, the coordinates and texture coordinates are both (0, 0, 0, 1), the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is (1, 1, 1, 1) and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value.







2.13 Colors and Coloring

Figures 2.8 and 2.9 diagram the processing of RGBA colors and color indices before rasterization. Incoming colors arrive in one of several formats. Table 2.6 summarizes the conversions that take place on R, G, B, and A components depending on which version of the **Color** command was invoked to specify the components. As a result of limited precision, some converted values will not be represented exactly. In color index mode, a single-valued color index is not mapped.

Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or color is used in further processing (the current color is the primary color, and the secondary color is (0, 0, 0, 0)). After lighting, RGBA colors are clamped to the range [0, 1]. A color index is converted to fixed-point and then its integer portion is masked (see section 2.13.6). After clamping or masking, a primitive may be *flatshaded*, indicating that all vertices of the primitive are to have the same color. Finally, if a primitive is clipped, then colors (and texture coordinates) must be computed at the vertices introduced or modified by clipping.

GL Type	Conversion
ubyte	$c/(2^8-1)$
byte	$(2c+1)/(2^8-1)$
ushort	$c/(2^{16}-1)$
short	$(2c+1)/(2^{16}-1)$
uint	$c/(2^{32}-1)$
int	$(2c+1)/(2^{32}-1)$
float	С
double	С

Table 2.6: Component conversions. Color, normal, and depth components, (c), are converted to an internal floating-point representation, (f), using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to components specified as parameters to GL commands and to components in pixel data. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2)

2.13.1 Lighting

GL lighting computes colors for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection of parameters that can include the vertex coordinates, the coordinates of one or more light sources, the current normal, and parameters defining the characteristics of the light sources and a current material. The following discussion assumes that the GL is in RGBA mode. (Color index lighting is described in section 2.13.5.)

Lighting may be in one of two states:

- 1. Lighting Off. In this state, the current color is assigned to the vertex primary color. The secondary color is (0, 0, 0, 0).
- 2. Lighting On. In this state, the vertex primary and secondary colors are computed from the current lighting parameters.

Lighting is turned on or off using the generic **Enable** or **Disable** commands with the symbolic value LIGHTING.

Lighting Operation

A lighting parameter is of one of five types: color, position, direction, real, or boolean. A color parameter consists of four floating-point values, one for each of R, G, B, and A, in that order. There are no restrictions on the allowable values for these parameters. A position parameter consists of four floating-point coordinates (x, y, z, and w) that specify a position in object coordinates (w may be zero, indicating a point at infinity in the directiongiven by <math>x, y, and z). A direction parameter consists of three floating-point coordinates (x, y, and z) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in Table 2.7. The result of a lighting computation is undefined if a value for a parameter is specified that is outside the range given for that parameter in the table.

There are *n* light sources, indexed by i = 0, ..., n-1. (*n* is an implementation dependent maximum that must be at least 8.) Note that the default values for \mathbf{d}_{cli} and \mathbf{s}_{cli} differ for i = 0 and i > 0.

Before specifying the way that lighting computes colors, we introduce operators and notation that simplify the expressions involved. If \mathbf{c}_1 and \mathbf{c}_2 are colors without alpha where $\mathbf{c}_1 = (r_1, g_1, b_1)$ and $\mathbf{c}_2 = (r_2, g_2, b_2)$, then define $\mathbf{c}_1 * \mathbf{c}_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$. Addition of colors is accomplished by addition of the components. Multiplication of colors by a scalar means multiplying each component by that scalar. If \mathbf{d}_1 and \mathbf{d}_2 are directions, then define

$$\mathbf{d}_1 \odot \mathbf{d}_2 = \max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0\}.$$

(Directions are taken to have three coordinates.) If \mathbf{P}_1 and \mathbf{P}_2 are (homogeneous, with four coordinates) points then let $\mathbf{P}_1\mathbf{P}_2$ be the unit vector that points from \mathbf{P}_1 to \mathbf{P}_2 . Note that if \mathbf{P}_2 has a zero w coordinate and \mathbf{P}_1 has non-zero w coordinate, then $\mathbf{P}_1\mathbf{P}_2$ is the unit vector corresponding to the direction specified by the x, y, and z coordinate of \mathbf{P}_2 ; if \mathbf{P}_1 has a zero w coordinate and \mathbf{P}_2 has a non-zero w coordinate then $\mathbf{P}_1\mathbf{P}_2$ is the unit vector that is the negative of that corresponding to the direction specified by \mathbf{P}_1 . If both \mathbf{P}_1 and \mathbf{P}_2 have zero w coordinates, then $\mathbf{P}_1\mathbf{P}_2$ is the unit vector obtained by normalizing the direction corresponding to $\mathbf{P}_2 - \mathbf{P}_1$.

If **d** is an arbitrary direction, then let **d** be the unit vector in **d**'s direction. Let $\|\mathbf{P}_1\mathbf{P}_2\|$ be the distance between \mathbf{P}_1 and \mathbf{P}_2 . Finally, let **V** be the point corresponding to the vertex being lit, and **n** be the corresponding normal. Let \mathbf{P}_e be the eyepoint ((0, 0, 0, 1) in eye coordinates).

Lighting produces two colors at a vertex: a primary color \mathbf{c}_{pri} and a secondary color \mathbf{c}_{sec} . The values of \mathbf{c}_{pri} and \mathbf{c}_{sec} depend on the light model

Parameter	Type	Default Value	Description		
Material Parameters					
\mathbf{a}_{cm}	color	$\left(0.2, 0.2, 0.2, 1.0 ight)$	ambient color of material		
\mathbf{d}_{cm}	color	$\left(0.8, 0.8, 0.8, 1.0 ight)$	diffuse color of material		
\mathbf{s}_{cm}	color	(0.0, 0.0, 0.0, 1.0)	specular color of material		
\mathbf{e}_{cm}	color	(0.0, 0.0, 0.0, 1.0)	emissive color of material		
s_{rm}	real	0.0	specular exponent (range:		
			[0.0, 128.0])		
a_m	real	0.0	ambient color index		
d_m	real	1.0	diffuse color index		
s_m	real	1.0	specular color index		
Light Source	e Parameter	S			
\mathbf{a}_{cli}	color	(0.0, 0.0, 0.0, 1.0)	ambient intensity of light i		
$\mathbf{d}_{cli}(i=0)$	color	(1.0, 1.0, 1.0, 1.0)	diffuse intensity of light 0		
$\mathbf{d}_{cli}(i>0)$	color	(0.0, 0.0, 0.0, 1.0)	diffuse intensity of light i		
$\mathbf{s}_{cli}(i=0)$	color	(1.0, 1.0, 1.0, 1.0)	specular intensity of light 0		
$\mathbf{s}_{cli}(i>0)$	color	(0.0, 0.0, 0.0, 1.0)	specular intensity of light i		
\mathbf{P}_{pli}	position	(0.0, 0.0, 1.0, 0.0)	position of light i		
\mathbf{s}_{dli}	direction	(0.0, 0.0, -1.0)	direction of spotlight for light		
			i		
s_{rli}	real	0.0	spotlight exponent for light i		
			(range: [0.0, 128.0])		
c_{rli}	real	180.0	spotlight cutoff angle for		
			light i (range: $[0.0, 90.0]$,		
			180.0)		
k_{0i}	real	1.0	constant attenuation factor		
			for light i (range: $[0.0,\infty)$)		
k_{1i}	real	0.0	linear attenuation factor for		
			light i (range: $[0.0,\infty)$)		
k_{2i}	real	0.0	quadratic attenuation factor		
			for light i (range: $[0.0,\infty)$)		
Lighting Mo	del Parame	ters	· · · · · · · · · · · · · · · · · · ·		
\mathbf{a}_{cs}	color	(0.2, 0.2, 0.2, 1.0)	ambient color of scene		
v_{bs}	boolean	FALSE	viewer assumed to be at		
			(0,0,0) in eye coordinates		
			(TRUE) or $(0, 0, \infty)$ (FALSE)		
c_{es}	enum	SINGLE_COLOR	controls computation of col-		
			ors		
t_{bs}	boolean	FALSE	use two-sided lighting mode		
	L	1			

Table 2.7: Summary of lighting parameters. The range of individual color components is $(-\infty, +\infty)$.

color control, $c_{es}.$ If $c_{es}=\texttt{SINGLE_COLOR},$ then the equations to compute \mathbf{c}_{pri} and \mathbf{c}_{sec} are

$$\begin{aligned} \mathbf{c}_{pri} &= \mathbf{e}_{cm} \\ &+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\ &+ \sum_{i=0}^{n-1} (att_i) (spot_i) \left[\mathbf{a}_{cm} * \mathbf{a}_{cli} \\ &+ (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli}) \mathbf{d}_{cm} * \mathbf{d}_{cli} \\ &+ (f_i) (\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli} \right] \\ \mathbf{c}_{sec} &= (0, 0, 0, 0) \end{aligned}$$

If $c_{es} = \texttt{SEPARATE_SPECULAR_COLOR}$, then

$$\begin{aligned} \mathbf{c}_{pri} &= \mathbf{e}_{cm} \\ &+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\ &+ \sum_{i=0}^{n-1} (att_i) (spot_i) \left[\mathbf{a}_{cm} * \mathbf{a}_{cli} \\ &+ (\mathbf{n} \odot \overline{\mathbf{VP}}_{pli}) \mathbf{d}_{cm} * \mathbf{d}_{cli} \right] \\ \mathbf{c}_{sec} &= \sum_{i=0}^{n-1} (att_i) (spot_i) (f_i) (\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli} \end{aligned}$$

where

$$f_{i} = \begin{cases} 1, & \mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli} \neq 0, \\ 0, & \text{otherwise}, \end{cases}$$
(2.2)

$$\mathbf{h}_{i} = \begin{cases} \overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_{e}, & v_{bs} = \text{TRUE}, \\ \overrightarrow{\mathbf{VP}}_{pli} + \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^{T}, & v_{bs} = \text{FALSE}, \end{cases}$$
(2.3)

$$att_{i} = \begin{cases} \frac{1}{k_{0i} + k_{1i} \|\mathbf{VP}_{pli}\| + k_{2i} \|\mathbf{VP}_{pli}\|^{2}}, & \text{if } \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases}$$
(2.4)

$$spot_{i} = \begin{cases} (\overline{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overline{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overline{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), (2.5) \\ 1.0, & c_{rli} = 180.0. \end{cases}$$

$$(2.6)$$

All computations are carried out in eye coordinates.

The value of A produced by lighting is the alpha value associated with \mathbf{d}_{cm} . A is always associated with the primary color \mathbf{c}_{pri} ; the alpha component of \mathbf{c}_{sec} is 0. Results of lighting are undefined if the w_e coordinate (w in eye coordinates) of \mathbf{V} is zero.

Lighting may operate in two-sided mode ($t_{bs} = \text{TRUE}$), in which a front color is computed with one set of material parameters (the front material) and a back color is computed with a second set of material parameters (the back material). This second computation replaces \mathbf{n} with $-\mathbf{n}$. If $t_{bs} = \text{FALSE}$, then the back color and front color are both assigned the color computed using the front material with \mathbf{n} .

The selection between back color and front color depends on the primitive of which the vertex being lit is a part. If the primitive is a point or a line segment, the front color is always selected. If it is a polygon, then the selection is based on the sign of the (clipped or unclipped) polygon's signed area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i$$
(2.7)

where x_w^i and y_w^i are the x and y window coordinates of the *i*th vertex of the *n*-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and $i \oplus 1$ is $(i + 1) \mod n$. The interpretation of the sign of this value is controlled with

void FrontFace(enum dir);

Setting dir to CCW (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that if $a \leq 0$, then the color of each vertex of the polygon becomes the back color computed for that vertex while if a > 0, then the front color is selected. If dir is CW, then a is replaced by -a in the above inequalities. This requires one bit of state; initially, it indicates CCW.

48

2.13.2 Lighting Parameter Specification

Lighting parameters are divided into three categories: material parameters, light source parameters, and lighting model parameters (see Table 2.7). Sets of lighting parameters are specified with

```
void Material{if}( enum face, enum pname, T param );
void Material{if}v( enum face, enum pname, T params );
void Light{if}( enum light, enum pname, T param );
void Light{if}v( enum light, enum pname, T params );
void LightModel{if}( enum pname, T param );
void LightModel{if}v( enum pname, T params );
```

pname is a symbolic constant indicating which parameter is to be set (see Table 2.8). In the vector versions of the commands, params is a pointer to a group of values to which to set the indicated parameter. The number of values pointed to depends on the parameter being set. In the non-vector versions, param is a value to which to set a single-valued parameter. (If param corresponds to a multi-valued parameter, the error INVALID_ENUM results.) For the Material command, face must be one of FRONT, BACK, or FRONT_AND_BACK, indicating that the property name of the front or back material, or both, respectively, should be set. In the case of Light, light is a symbolic constant of the form LIGHT*i*, indicating that light *i* is to have the specified parameter set. The constants obey LIGHT*i* = LIGHTO + *i*.

Table 2.8 gives, for each of the three parameter groups, the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each. Color parameters specified with **Material** and **Light** are converted to floating-point values (if specified as integers) as indicated in Table 2.6 for signed integers. The error INVALID_VALUE occurs if a specified lighting parameter lies outside the allowable range given in Table 2.7. (The symbol " ∞ " indicates the maximum representable magnitude for the indicated type.)

The current model-view matrix is applied to the position parameter indicated with **Light** for a particular light source when that position is specified. These transformed values are the values used in the lighting equation.

The spotlight direction is transformed when it is specified using only the upper leftmost 3x3 portion of the model-view matrix. That is, if \mathbf{M}_u is the upper left 3x3 matrix taken from the current model-view matrix M, then

Parameter	Name	Number of values				
Material Pa	Material Parameters (Material)					
\mathbf{a}_{cm}	AMBIENT	4				
\mathbf{d}_{cm}	DIFFUSE	4				
$\mathbf{a}_{cm}, \mathbf{d}_{cm}$	AMBIENT_AND_DIFFUSE	4				
\mathbf{s}_{cm}	SPECULAR	4				
\mathbf{e}_{cm}	EMISSION	4				
s_{rm}	SHININESS	1				
a_m, d_m, s_m	COLOR_INDEXES	3				
Light Source	e Parameters (\mathbf{Light})					
\mathbf{a}_{cli}	AMBIENT	4				
\mathbf{d}_{cli}	DIFFUSE	4				
\mathbf{s}_{cli}	SPECULAR	4				
\mathbf{P}_{pli}	POSITION	4				
\mathbf{s}_{dli}	SPOT_DIRECTION	3				
s_{rli}	SPOT_EXPONENT	1				
c_{rli}	SPOT_CUTOFF	1				
k_0	CONSTANT_ATTENUATION	1				
k_1	LINEAR_ATTENUATION	1				
k_2	QUADRATIC_ATTENUATION	1				
Lighting Model Parameters (LightModel)						
\mathbf{a}_{cs}	LIGHT_MODEL_AMBIENT	4				
v_{bs}	LIGHT_MODEL_LOCAL_VIEWER	1				
t_{bs}	LIGHT_MODEL_TWO_SIDE	1				
c_{es}	LIGHT_MODEL_COLOR_CONTROL	1				

Table 2.8: Correspondence of lighting parameter symbols to names. AMBIENT_AND_DIFFUSE is used to set \mathbf{a}_{cm} and \mathbf{d}_{cm} to the same value.

the spotlight direction

$$\begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

is transformed to

$$\begin{pmatrix} d'_x \\ d'_y \\ d'_z \end{pmatrix} = M_u \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}.$$

An individual light is enabled or disabled by calling **Enable** or **Disable** with the symbolic value LIGHTi (i is in the range 0 to n - 1, where n is the implementation-dependent number of lights). If light i is disabled, the ith term in the lighting equation is effectively removed from the summation.

2.13.3 ColorMaterial

It is possible to attach one or more material properties to the current color, so that they continuously track its component values. This behavior is enabled and disabled by calling **Enable** or **Disable** with the symbolic value **COLOR_MATERIAL**.

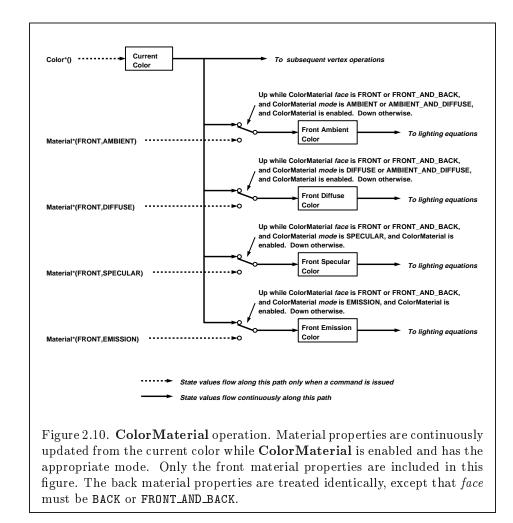
The command that controls which of these modes is selected is

void ColorMaterial(enum face, enum mode);

face is one of FRONT, BACK, or FRONT_AND_BACK, indicating whether the front material, back material, or both are affected by the current color. mode is one of EMISSION, AMBIENT, DIFFUSE, SPECULAR, or AMBIENT_AND_DIFFUSE and specifies which material property or properties track the current color. If mode is EMISSION, AMBIENT, DIFFUSE, or SPECULAR, then the value of \mathbf{e}_{cm} , \mathbf{a}_{cm} , \mathbf{d}_{cm} or \mathbf{s}_{cm} , respectively, will track the current color. If mode is AMBIENT_AND_DIFFUSE, both \mathbf{a}_{cm} and \mathbf{d}_{cm} track the current color. The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when ColorMaterial is not currently enabled to override that particular value. When COLOR_MATERIAL is enabled, the indicated parameter or parameters always track the current color. For instance, calling

ColorMaterial (FRONT, AMBIENT)

while COLOR_MATERIAL is enabled sets the front material \mathbf{a}_{cm} to the value of the current color.



2.13.4 Lighting State

The state required for lighting consists of all of the lighting parameters (front and back material parameters, lighting model parameters, and at least 8 sets of light parameters), a bit indicating whether a back color distinct from the front color should be computed, at least 8 bits to indicate which lights are enabled, a five-valued variable indicating the current **ColorMaterial** mode, a bit indicating whether or not **COLOR_MATERIAL** is enabled, and a single bit to indicate whether lighting is enabled or disabled. In the initial state, all lighting parameters have their default values. Back color evaluation does not take place, **ColorMaterial** is **FRONT_AND_BACK** and **AMBIENT_AND_DIFFUSE**, and both lighting and **COLOR_MATERIAL** are disabled.

2.13.5 Color Index Lighting

A simplified lighting computation applies in color index mode that uses many of the parameters controlling RGBA lighting, but none of the RGBA material parameters. First, the RGBA diffuse and specular intensities of light i (\mathbf{d}_{cli} and \mathbf{s}_{cli} , respectively) determine color index diffuse and specular light intensities, d_{li} and s_{li} from

$$d_{li} = (.30)R(\mathbf{d}_{cli}) + (.59)G(\mathbf{d}_{cli}) + (.11)B(\mathbf{d}_{cli})$$

and

$$s_{li} = (.30)R(\mathbf{s}_{cli}) + (.59)G(\mathbf{s}_{cli}) + (.11)B(\mathbf{s}_{cli}).$$

 $R(\mathbf{x})$ indicates the R component of the color \mathbf{x} and similarly for $G(\mathbf{x})$ and $B(\mathbf{x})$.

Next, let

$$s = \sum_{i=0}^{n} (att_i)(spot_i)(s_{li})(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}$$

where att_i and $spot_i$ are given by equations 2.4 and 2.5, respectively, and f_i and $\hat{\mathbf{h}}_i$ are given by equations 2.2 and 2.3, respectively. Let $s' = \min\{s, 1\}$. Finally, let

$$d = \sum_{i=0}^{n} (att_i)(spot_i)(d_{li})(\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli}).$$

Then color index lighting produces a value c, given by

$$c = a_m + d(1 - s')(d_m - a_m) + s'(s_m - a_m).$$

The final color index is

$$c' = \min\{c, s_m\}.$$

The values a_m , d_m and s_m are material properties described in Tables 2.7 and 2.8. Any ambient light intensities are incorporated into a_m . As with RGBA lighting, disabled lights cause the corresponding terms from the summations to be omitted. The interpretation of t_{bs} and the calculation of front and back colors is carried out as has already been described for RGBA lighting.

The values a_m , d_m , and s_m are set with **Material** using a *pname* of **COLOR_INDEXES**. Their initial values are 0, 1, and 1, respectively. The additional state consists of three floating-point values. These values have no effect on RGBA lighting.

2.13.6 Clamping or Masking

After lighting (whether enabled or not), all components of both primary and secondary colors are clamped to the range [0, 1].

For a color index, the index is first converted to fixed-point with an unspecified number of bits to the right of the binary point; the nearest fixed-point value is selected. Then, the bits to the right of the binary point are left alone while the integer portion is masked (bitwise ANDed) with $2^n - 1$, where n is the number of bits in a color in the color index buffer (buffers are discussed in chapter 4).

2.13.7 Flatshading

A primitive may be *flatshaded*, meaning that all vertices of the primitive are assigned the same color index or the same primary and secondary colors. These colors are the colors of the vertex that spawned the primitive. For a point, these are the colors associated with the point. For a line segment, they are the colors of the second (final) vertex of the segment. For a polygon, they come from a selected vertex depending on how the polygon was generated. Table 2.9 summarizes the possibilities.

Flatshading is controlled by

```
void ShadeModel( enum mode );
```

mode value must be either of the symbolic constants SMOOTH or FLAT. If mode is SMOOTH (the initial state), vertex colors are treated individually. If mode is FLAT, flatshading is turned on. ShadeModel thus requires one bit of state.

Primitive type of polygon i	Vertex
single polygon $(i \equiv 1)$	1
triangle strip	i+2
triangle fan	i+2
independent triangle	3i
quad strip	2i + 2
independent quad	4i

Table 2.9: Polygon flatshading color selection. The colors used for flatshading the *i*th polygon generated by the indicated **Begin/End** type are derived from the current color (if lighting is disabled) in effect when the indicated vertex is specified. If lighting is enabled, the colors are produced by lighting the indicated vertex. Vertices are numbered 1 through n, where n is the number of vertices between the **Begin/End** pair.

2.13.8 Color and Texture Coordinate Clipping

After lighting, clamping or masking and possible flatshading, colors are clipped. Those colors associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the colors assigned to vertices produced by clipping are clipped colors.

Let the colors assigned to the two vertices \mathbf{P}_1 and \mathbf{P}_2 of an unclipped edge be \mathbf{c}_1 and \mathbf{c}_2 . The value of t (section 2.11) for a clipped point \mathbf{P} is used to obtain the color associated with \mathbf{P} as

$$\mathbf{c} = t\mathbf{c}_1 + (1-t)\mathbf{c}_2.$$

(For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For an RGBA color, it means multiplying each of R, G, B, and A by the scalar. Both primary and secondary colors are treated in the same fashion.) Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Color clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

Texture coordinates must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

2.13.9 Final Color Processing

For an RGBA color, each color component (which lies in [0, 1]) is converted (by rounding to nearest) to a fixed-point value with m bits. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones). m must be at least as large as the number of bits in the corresponding component of the framebuffer. m must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer. A color index is converted (by rounding to nearest) to a fixed-point value with at least as many bits as there are in the color index portion of the framebuffer.

Because a number of the form $k/(2^m-1)$ may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of RGBA components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and one of **Colorub**, **Colorus**, or **Colorui** was used to specify that color. When these conditions are satisfied, an RGBA component must convert to a value that matches the component as specified in the **Color** command: if m is less than the number of bits b with which the component was specified, then the converted value must equal the most significant m bits of the specified value; otherwise, the most significant b bits of the converted value must equal the specified value.

Chapter 3

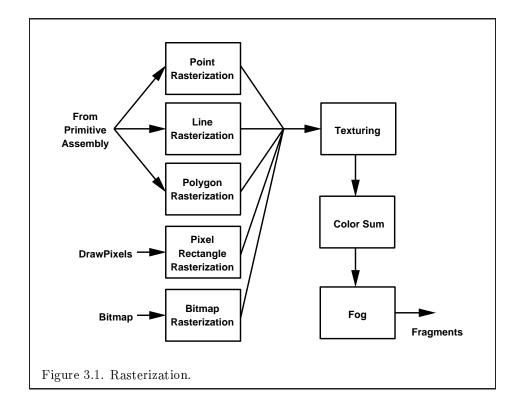
Rasterization

Rasterization is the process by which a primitive is converted to a twodimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a color and a depth value to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process.

A grid square along with its parameters of assigned color, z (depth), and texture coordinates is called a *fragment*; the parameters are collectively dubbed the fragment's *associated data*. A fragment is located by its lowerleft corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment's *center*, which is offset by (1/2, 1/2) from its lowerleft corner (and so lies on half-integer coordinates).

Grid squares need not actually be square in the GL. Rasterization rules are not affected by the actual aspect ratio of the grid squares. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing.

Several factors affect rasterization. Lines and polygons may be stippled. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.



3.1 Invariance

Consider a primitive p' obtained by translating a primitive p through an offset (x, y) in window coordinates, where x and y are integers. As long as neither p' nor p is clipped, it must be the case that each fragment f' produced from p' is identical to a corresponding fragment f from p except that the center of f' is offset by (x, y) from the center of f.

3.2 Antialiasing

Antialiasing of a point, line, or polygon is effected in one of two ways depending on whether the GL is in RGBA or color index mode.

In RGBA mode, the R, G, and B values of the rasterized fragment are left unaffected, but the A value is multiplied by a floating-point value in the range [0, 1] that describes a fragment's screen pixel coverage. The per-fragment stage of the GL can be set up to use the A value to blend the incoming fragment with the corresponding pixel already present in the framebuffer.

In color index mode, the least significant b bits (to the left of the binary point) of the color index are used for antialiasing; $b = \min\{4, m\}$, where m is the number of bits in the color index portion of the framebuffer. The antialiasing process sets these b bits based on the fragment's coverage value: the bits are set to zero for no coverage and to all ones for complete coverage.

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take into account perceptual issues as well as characteristics of the monitor on which the contents of the framebuffer are displayed. Such details cannot be addressed within the scope of this document. Further, the coverage value computed for a fragment of some primitive may depend on the primitive's relationship to a number of grid squares neighboring the one corresponding to the fragment, and not just on the fragment's grid square. Another consideration is that accurate calculation of coverage values may be computationally expensive; consequently we allow a given GL implementation to approximate true coverage values by using a fast but not entirely accurate coverage computation.

In light of these considerations, we chose to specify the behavior of exact antialiasing in the prototypical case that each displayed pixel is a perfect square of uniform intensity. The square is called a *fragment square* and has lower left corner (x, y) and upper right corner (x + 1, y + 1). We recognize

that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, well-defined model.

A GL implementation may use other methods to perform antialiasing, subject to the following conditions:

- 1. If f_1 and f_2 are two fragments, and the portion of f_1 covered by some primitive is a subset of the corresponding portion of f_2 covered by the primitive, then the coverage computed for f_1 must be less than or equal to that computed for f_2 .
- 2. The coverage computation for a fragment f must be local: it may depend only on f's relationship to the boundary of the primitive being rasterized. It may not depend on f's x and y coordinates.

Another property that is desirable, but not required, is:

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.6), allowing a user to make an image quality versus speed tradeoff.

3.3 Points

The rasterization of points is controlled with

void PointSize(float size);

size specifies the width or diameter of a point. The default value is 1.0. A value less than or equal to zero results in the error INVALID_VALUE.

Point antialiasing is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant POINT_SMOOTH. The default state is for point antialiasing to be disabled.

In the default state, a point is rasterized by truncating its x_w and y_w coordinates (recall that the subscripts indicate that these are x and y window coordinates) to integers. This (x, y) address, along with data derived from the data associated with the vertex corresponding to the point, is sent as a single fragment to the per-fragment stage of the GL.

3.3. POINTS

The effect of a point width other than 1.0 depends on the state of point antialiasing. If antialiasing is disabled, the actual width is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased point width. This implementation-dependent value must be no less than the implementationdependent maximum antialiased point width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1. If the resulting width is odd, then the point

$$(x,y) = (\lfloor x_w \rfloor + \frac{1}{2}, \lfloor y_w \rfloor + \frac{1}{2})$$

is computed from the vertex's x_w and y_w , and a square grid of the odd width centered at (x, y) defines the centers of the rasterized fragments (recall that fragment centers lie at half-integer window coordinate values). If the width is even, then the center point is

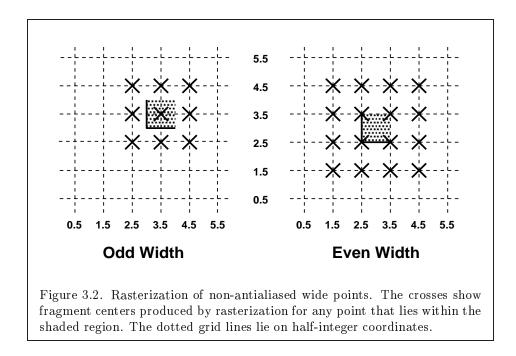
$$(x,y) = (\lfloor x_w + \frac{1}{2} \rfloor, \lfloor y_w + \frac{1}{2} \rfloor);$$

the rasterized fragment centers are the half-integer window coordinate values within the square of the even width centered on (x, y). See figure 3.2.

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point, with texture coordinates s, t, and r replaced with s/q, t/q, and r/q, respectively. If q is less than or equal to zero, the results are undefined.

If antialiasing is enabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered at the point's (x_w, y_w) (figure 3.3). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding fragment square (but see section 3.2). This value is saved and used in the final step of rasterization (section 3.11). The data associated with each fragment are otherwise the data associated with the point being rasterized, with texture coordinates s, t, and r replaced with s/q, t/q, and r/q, respectively. If q is less than or equal to zero, the results are undefined.

Not all widths need be supported when point antialiasing is on, but the width 1.0 must be provided. If an unsupported width is requested, the nearest supported width is used instead. The range of supported widths and the width of evenly-spaced gradations within that range are implementation dependent. The range and gradations may be obtained using the query



mechanism described in Chapter 6. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths $0.1, 0.2, \ldots, 1.9, 2.0$ are supported.

3.3.1 Point Rasterization State

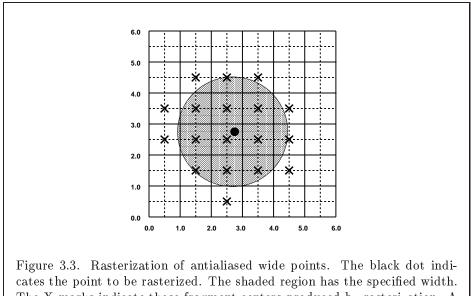
The state required to control point rasterization consists of the floating-point point width and a bit indicating whether or not antialiasing is enabled.

3.4 Line Segments

A line segment results from a line strip **Begin/End** object, a line loop, or a series of separate line segments. Line segment rasterization is controlled by several variables. Line width, which may be set by calling

```
void LineWidth( float width );
```

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0. Values less than or equal



cates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region that covers the corresponding fragment square. Solid lines lie on integer coordinates. to 0.0 generate the error INVALID_VALUE. Antialiasing is controlled with Enable and Disable using the symbolic constant LINE_SMOOTH. Finally, line segments may be stippled. Stippling is controlled by a GL command that sets a *stipple pattern* (see below).

3.4.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either x-major or y-major. x-major line segments have slope in the closed interval [-1, 1]; all other line segments are y-major (slope is determined by the segment's endpoints). We shall specify rasterization only for x-major segments except in cases where the modifications for y-major segments are not self-evident.

Ideally, the GL uses a "diamond-exit" rule to determine those fragments that are produced by rasterizing a line segment. For each fragment f with center at window coordinates x_f and y_f , define a diamond-shaped region that is the intersection of four half planes:

$$R_{f} = \{ (x, y) \mid |x - x_{f}| + |y - y_{f}| < 1/2. \}$$

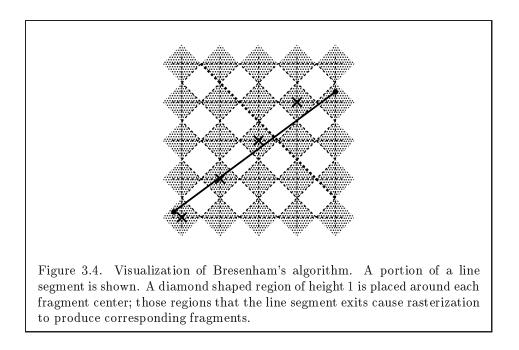
Essentially, a line segment starting at \mathbf{p}_a and ending at \mathbf{p}_b produces those fragments f for which the segment intersects R_f , except if \mathbf{p}_b is contained in R_f . See figure 3.4.

To avoid difficulties when an endpoint lies on a boundary of R_f we (in principle) perturb the supplied endpoints by a tiny amount. Let \mathbf{p}_a and \mathbf{p}_b have window coordinates (x_a, y_a) and (x_b, y_b) , respectively. Obtain the perturbed endpoints \mathbf{p}'_a given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ and \mathbf{p}'_b given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line segment starting at \mathbf{p}_a and ending at \mathbf{p}_b produces those fragments f for which the segment starting at \mathbf{p}'_a and ending on \mathbf{p}'_b intersects R_f , except if \mathbf{p}'_b is contained in R_f . ϵ is chosen to be so small that rasterizing the line segment produces the same fragments when δ is substituted for ϵ for any $0 < \delta \leq \epsilon$.

When \mathbf{p}_a and \mathbf{p}_b lie on fragment centers, this characterization of fragments reduces to Bresenham's algorithm with one modification: lines produced in this description are "half-open," meaning that the final fragment (corresponding to \mathbf{p}_b) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham's algorithm).

Because the initial and final conditions of the diamond-exit rule may be difficult to implement, other line segment rasterization algorithms are allowed, subject to the following rules:

3.4. LINE SEGMENTS



- 1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either x or y window coordinates from a corresponding fragment produced by the diamond-exit rule.
- 2. The total number of fragments produced by the algorithm may differ from that produced by the diamond-exit rule by no more than one.
- 3. For an x-major line, no two fragments may be produced that lie in the same window-coordinate column (for a y-major line, no two fragments may appear in the same row).
- 4. If two line segments share a common endpoint, and both segments are either x-major (both left-to-right or both right-to-left) or y-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

Next we must specify how the data associated with each rasterized fragment are obtained. Let the window coordinates of a produced fragment center be given by $\mathbf{p}_r = (x_d, y_d)$ and let $\mathbf{p}_a = (x_a, y_a)$ and $\mathbf{p}_b = (x_b, y_b)$. Set

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}.$$
(3.1)

(Note that t = 0 at \mathbf{p}_a and t = 1 at \mathbf{p}_b .) The value of an associated datum f for the fragment, whether it be R, G, B, or A (in RGBA mode) or a color index (in color index mode), or the s, t, or r texture coordinate (the depth value, window z, must be found using equation 3.3, below), is found as

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)\alpha_a/w_a + t\alpha_b/w_b}$$
(3.2)

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively. $\alpha_a = \alpha_b = 1$ for all data except texture coordinates, in which case $\alpha_a = q_a$ and $\alpha_b = q_b$ $(q_a \text{ and } q_b \text{ are the homogeneous texture coordinates at the starting and end$ ing endpoints of the segment; results are undefined if either of these is lessthan or equal to 0). Note that linear interpolation would use

$$f = (1-t)f_a/\alpha_a + tf_b/\alpha_b.$$
(3.3)

The reason that this formula is incorrect (except for the depth value) is that it interpolates a datum in window space, which may be distorted by perspective. What is actually desired is to find the corresponding value when interpolated in clip space, which equation 3.2 does. A GL implementation may choose to approximate equation 3.2 with 3.3, but this will normally lead to unacceptable distortion effects when interpolating texture coordinates.

3.4.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one using the default line stipple of $FFFF_{16}$. We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

Line Stipple

The command

void LineStipple(int factor, ushort pattern);

defines a *line stipple*. *pattern* is an unsigned short integer. The *line stipple* is taken from the lowest order 16 bits of *pattern*. It determines those fragments that are to be drawn when the line is rasterized. *factor* is a count that is used to modify the effective line stipple by causing each bit in *line stipple* to be used *factor* times. *factor* is clamped to the range [1, 256]. Line stippling may be enabled or disabled using **Enable** or **Disable** with the constant LINE_STIPPLE. When disabled, it is as if the line stipple has its default value.

Line stippling masks certain fragments that are produced by rasterization so that they are not sent to the per-fragment stage of the GL. The masking is achieved using three parameters: the 16-bit line stipple p, the line repeat count r, and an integer stipple counter s. Let

$$b = \lfloor s/r \rfloor \mod 16,$$

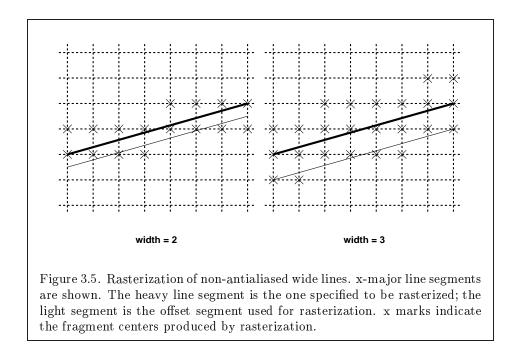
Then a fragment is produced if the *b*th bit of p is 1, and not produced otherwise. The bits of p are numbered with 0 being the least significant and 15 being the most significant. The initial value of s is zero; s is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). s is reset to 0 whenever a **Begin** occurs, and before every line segment in a group of independent segments (as specified when **Begin** is invoked with LINES).

If the line segment has been clipped, then the value of s at the beginning of the line segment is indeterminate.

Wide Lines

The actual width of non-antialiased lines is determined by rounding the supplied width to the nearest integer, then clamping it to the implementationdependent maximum non-antialiased line width. This implementationdependent value must be no less than the implementation-dependent maximum antialiased line width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1.

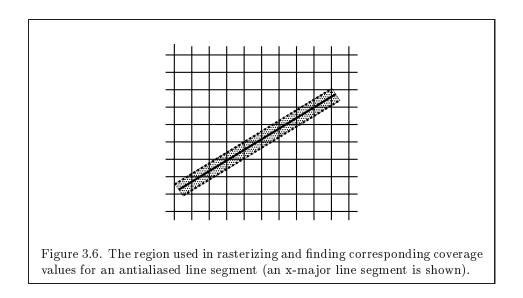
Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an x-major line, the minor direction is y, and for a y-major line, the minor direction is x) and replicating fragments in the minor direction (see figure 3.5). Let w be the width rounded to the nearest integer (if w = 0, then it is as if w = 1). If the line segment has endpoints given by (x_0, y_0) and (x_1, y_1) in window coordinates, the segment with endpoints $(x_0, y_0 - (w - 1)/2)$ and $(x_1, y_1 - (w - 1)/2)$ is rasterized, but



instead of a single fragment, a column of fragments of height w (a row of fragments of length w for a y-major segment) is produced at each x (y for y-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates. The whole column is not produced if the stipple bit for the column's x location is zero; otherwise, the whole column is produced.

Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of the rectangle with the fragment square (see figure 3.6; see also section 3.2). Equation 3.2 is used to compute associated data values just as with non-antialiased lines; equation 3.1 is used to find the value of t for each fragment whose square is intersected by the line segment's rectangle. Not all widths need be sup-



ported for line segment antialiasing, but width 1.0 antialiased segments must be provided. As with the point width, a GL implementation may be queried for the range and number of gradations of available antialiased line widths.

For purposes of antialiasing, a stippled line is considered to be a sequence of contiguous rectangles centered on the line segment. Each rectangle has width equal to the current line width and length equal to 1 pixel (except the last, which may be shorter). These rectangles are numbered from 0 to n, starting with the rectangle incident on the starting endpoint of the segment. Each of these rectangles is either eliminated or produced according to the procedure given under **Line Stipple**, above, where "fragment" is replaced with "rectangle." Each rectangle so produced is rasterized as if it were an antialiased polygon, described below (but culling, non-default settings of **PolygonMode**, and polygon stippling are not applied).

3.4.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width, a 16-bit line stipple, the line stipple repeat count, a bit indicating whether stippling is enabled or disabled, and a bit indicating whether line antialiasing is on or off. In addition, during rasterization, an integer stipple counter must be maintained to implement line stippling. The initial value of the line width is 1.0. The initial value of the line stipple is FFF_{16} (a stipple of all ones). The initial value of the line stipple repeat count is one.

The initial state of line stippling is disabled. The initial state of line segment antialiasing is disabled.

3.5 Polygons

A polygon results from a polygon **Begin/End** object, a triangle resulting from a triangle strip, triangle fan, or series of separate triangles, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or a **Rect** command. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is controlled with **Enable** and **Disable** with the symbolic constant POLYGON_SMOOTH. The analog to line segment stippling for polygons is polygon stippling, described below.

3.5.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back facing* or *front facing*. This determination is made by examining the sign of the area computed by equation 2.7 of section 2.13.1 (including the possible reversal of this sign as indicated by the last call to **FrontFace**). If this sign is positive, the polygon is frontfacing; otherwise, it is back facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

void CullFace(enum mode);

mode is a symbolic constant: one of FRONT, BACK or FRONT_AND_BACK. Culling is enabled or disabled with **Enable** or **Disable** using the symbolic constant CULL_FACE. Front facing polygons are rasterized if either culling is disabled or the **CullFace** mode is BACK while back facing polygons are rasterized only if either culling is disabled or the **CullFace** mode is FRONT. The initial setting of the **CullFace** mode is BACK. Initially, culling is disabled.

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

3.5. POLYGONS

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, a, b, and c, each in the range [0, 1], with a + b + c = 1. These coordinates uniquely specify any point p within the triangle or on the triangle's boundary as

$$p = ap_a + bp_b + cp_c,$$

where p_a , p_b , and p_c are the vertices of the triangle. a, b, and c can be found as

$$a=rac{\mathrm{A}(pp_bp_c)}{\mathrm{A}(p_ap_bp_c)}, \quad b=rac{\mathrm{A}(pp_ap_c)}{\mathrm{A}(p_ap_bp_c)}, \quad c=rac{\mathrm{A}(pp_ap_b)}{\mathrm{A}(p_ap_bp_c)},$$

where A(lmn) denotes the area in window coordinates of the triangle with vertices l, m, and n.

Denote a datum at p_a , p_b , or p_c as f_a , f_b , or f_c , respectively. Then the value f of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a\alpha_a/w_a + b\alpha_b/w_b + c\alpha_c/w_c}$$
(3.4)

where w_a , w_b and w_c are the clip w coordinates of p_a , p_b , and p_c , respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. $\alpha_a = \alpha_b = \alpha_c = 1$ except for texture s, t, and r coordinates, for which $\alpha_a = q_a$, $\alpha_b = q_b$, and $\alpha_c = q_c$ (if any of q_a , q_b , or q_c are less than or equal to zero, results are undefined). a, b, and c must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center.

Just as with line segment rasterization, equation 3.4 may be approximated by

$$f = af_a/\alpha_a + bf_b/\alpha_b + cf_c/\alpha_c;$$

this may yield acceptable results for color values (it *must* be used for depth values), but will normally lead to unacceptable distortion effects if used for texture coordinates.

For a polygon with more than three edges, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

$$f = \sum_{i=1}^{n} a_i f_i$$

where n is the number of vertices in the polygon, f_i is the value of the f at vertex i; for each $i \ 0 \le a_i \le 1$ and $\sum_{i=1}^n a_i = 1$. The values of the a_i may differ from fragment to fragment, but at vertex $i, a_j = 0, j \ne i$ and $a_i = 1$.

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 3.4 should be iterated independently and a division performed for each fragment).

3.5.2 Stippling

Polygon stippling works much the same way as line stippling, masking out certain fragments produced by rasterization so that they are not sent to the next stage of the GL. This is the case regardless of the state of polygon antialiasing. Stippling is controlled with

```
void PolygonStipple( ubyte *pattern );
```

pattern is a pointer to memory into which a 32×32 pattern is packed. The pattern is unpacked from memory according to the procedure given in section 3.6.4 for **DrawPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were **BITMAP**, and the *format* were **COLOR_INDEX**. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones.

If x_w and y_w are the window coordinates of a rasterized polygon fragment, then that fragment is sent to the next stage of the GL if and only if the bit of the pattern ($x_w \mod 32$, $y_w \mod 32$) is 1.

Polygon stippling may be enabled or disabled with **Enable** or **Disable** using the constant POLYGON_STIPPLE. When disabled, it is as if the stipple pattern were all ones.

3.5.3 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is computed at each such fragment, and this value is saved to be applied as described in section 3.11. An associated datum is assigned to a fragment by integrating the datum's value over the region of the intersection of the fragment square with the polygon's interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment's center may be used instead of integrating the value across the fragment.

Polygon stippling operates in the same way whether polygon antialiasing is enabled or not. The polygon point sampling rule defined in section 3.5.1, however, is not enforced for antialiased polygons.

3.5.4 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using

```
void PolygonMode( enum face, enum mode );
```

face is one of FRONT, BACK, or FRONT_AND_BACK, indicating that the rasterizing method described by *mode* replaces the rasterizing method for front facing polygons, back facing polygons, or both front and back facing polygons, respectively. mode is one of the symbolic constants POINT, LINE, or FILL. Calling PolygonMode with POINT causes certain vertices of a polygon to be treated, for rasterization purposes, just as if they were enclosed within a Begin(POINT) and End pair. The vertices selected for this treatment are those that have been tagged as having a polygon boundary edge beginning on them (see section 2.6.2). LINE causes edges that are tagged as boundary to be rasterized as line segments. (The line stipple counter is reset at the beginning of the first rasterized edge of the polygon, but not for subsequent edges.) FILL is the default mode of polygon rasterization, corresponding to the description in sections 3.5.1, 3.5.2, and 3.5.3. Note that these modes affect only the final rasterization of polygons: in particular, a polygon's vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

Polygon antialiasing applies only to the FILL state of **PolygonMode**. For **POINT** or **LINE**, point antialiasing or line segment antialiasing, respectively, apply.

3.5.5 Depth Offset

The depth values of all fragments generated by the rasterization of a polygon may be offset by a single value that is computed for that polygon. The function that determines this value is specified by calling

void PolygonOffset(float factor, float units);

factor scales the maximum depth slope of the polygon, and *units* scales an implementation dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the polygon offset value. Both *factor* and *units* may be either positive or negative.

The maximum depth slope m of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{3.5}$$

where (x_w, y_w, z_w) is a point on the triangle. m may be approximated as

$$m = \max\left\{ \left| \frac{\partial z_w}{\partial x_w} \right|, \left| \frac{\partial z_w}{\partial y_w} \right| \right\}.$$
(3.6)

If the polygon has more than three vertices, one or more values of m may be used during rasterization. Each may take any value in the range [min,max], where min and max are the smallest and largest values obtained by evaluating Equation 3.5 or Equation 3.6 for the triangles formed by all three-vertex combinations.

The minimum resolvable difference r is an implementation constant. It is the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but z_w values that differ by r, will have distinct depth values.

The offset value o for a polygon is

$$o = m * factor + r * units.$$
(3.7)

m is computed as described above, as a function of depth values in the range [0,1], and o is applied to depth values in the same range.

Boolean state values POLYGON_OFFSET_POINT, POLYGON_OFFSET_LINE, and POLYGON_OFFSET_FILL determine whether *o* is applied during the rasterization of polygons in POINT, LINE, and FILL modes. These boolean state values are enabled and disabled as argument values to the commands **Enable** and **Disable**. If POLYGON_OFFSET_POINT is enabled, *o* is added to the depth value of each fragment produced by the rasterization of a polygon in POINT mode. Likewise, if POLYGON_OFFSET_LINE or POLYGON_OFFSET_FILL is enabled, *o* is added to the depth value of each fragment produced by the rasterization of a polygon in LINE or FILL modes, respectively.

Fragment depth values are always limited to the range [0,1], either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygon.

3.5.6 Polygon Rasterization State

The state required for polygon rasterization consists of a polygon stipple pattern, whether stippling is enabled or disabled, the current state of polygon antialiasing (enabled or disabled), the current values of the **PolygonMode** setting for each of front and back facing polygons, whether point, line, and fill mode polygon offsets are enabled or disabled, and the factor and bias values of the polygon offset equation. The initial stipple pattern is all ones; initially stippling is disabled. The initial setting of polygon antialiasing is disabled. The initial state for **PolygonMode** is **FILL** for both front and back facing polygons. The initial polygon offset factor and bias values are both 0; initially polygon offset is disabled for all modes.

3.6 Pixel Rectangles

Rectangles of color, depth, and certain other values may be converted to fragments using the **DrawPixels** command (described in section 3.6.4). Some of the parameters and operations governing the operation of **Draw-Pixels** are shared by **ReadPixels** (used to obtain pixel values from the framebuffer) and **CopyPixels** (used to copy pixels from one framebuffer location to another); the discussion of **ReadPixels** and **CopyPixels**, however, is deferred until Chapter 4 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to **DrawPixels** also pertain to **ReadPixels** or **CopyPixels**.

A number of parameters control the encoding of pixels in client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying). These parameters are set with three commands: **PixelStore**, **PixelTransfer**, and **PixelMap**.

3.6.1 Pixel Storage Modes

Pixel storage modes affect the operation of **DrawPixels** and **ReadPixels** (as well as other commands; see sections 3.5.2, 3.7, and 3.8) when one of

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	$[0,\infty)$
UNPACK_SKIP_ROWS	integer	0	$[0,\infty)$
UNPACK_SKIP_PIXELS	integer	0	$[0,\infty)$
UNPACK_ALIGNMENT	integer	4	1,2,4,8
UNPACK_IMAGE_HEIGHT	integer	0	$[0,\infty)$
UNPACK_SKIP_IMAGES	integer	0	$[0,\infty)$

Table 3.1: **PixelStore** parameters pertaining to one or more of **DrawPixels**, **TexImage1D**, **TexImage2D**, and **TexImage3D**.

these commands is issued. This may differ from the time that the command is executed if the command is placed in a display list (see section 5.4). Pixel storage modes are set with

void PixelStore{if}(enum pname, T param);

pname is a symbolic constant indicating a parameter to be set, and param is the value to set it to. Table 3.1 summarizes the pixel storage parameters, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error INVALID_VALUE.

The version of **PixelStore** that takes a floating-point value may be used to set any type of parameter; if the parameter is boolean, then it is set to FALSE if the passed value is 0.0 and TRUE otherwise, while if the parameter is an integer, then the passed value is rounded to the nearest integer. The integer version of the command may also be used to set any type of parameter; if the parameter is boolean, then it is set to FALSE if the passed value is 0 and TRUE otherwise, while if the passed value is 0 and TRUE otherwise, while if the parameter is a floating-point value, then the passed value is converted to floating-point.

3.6.2 The Imaging Subset

Some pixel transfer and per-fragment operations are only made available in GL implementations which incorporate the optional *imaging subset*. The imaging subset includes both new commands, and new enumerants allowed as parameters to existing commands. If the subset is supported, *all* of these

calls and enumerants must be implemented as described later in the GL specification. If the subset is not supported, calling any of the new commands generates the error INVALID_OPERATION, and using any of the new enumerants generates the error INVALID_ENUM.

The individual operations available only in the imaging subset are described in section 3.6.3, except for blending features, which are described in chapter 4. Imaging subset operations include:

- 1. Color tables, including all commands and enumerants described in subsections Color Table Specification, Alternate Color Table Specification Commands, Color Table State and Proxy State, Color Table Lookup, Post Convolution Color Table Lookup, and Post Color Matrix Color Table Lookup, as well as the query commands described in section 6.1.7.
- 2. Convolution, including all commands and enumerants described in subsections Convolution Filter Specification, Alternate Convolution Filter Specification Commands, and Convolution, as well as the query commands described in section 6.1.8.
- 3. Color matrix, including all commands and enumerants described in subsections Color Matrix Specification and Color Matrix Transformation, as well as the simple query commands described in section 6.1.6.
- Histogram and minmax, including all commands and enumerants described in subsections Histogram Table Specification, Histogram State and Proxy State, Histogram, Minmax Table Specification, and Minmax, as well as the query commands described in section 6.1.9 and section 6.1.10.
- 5. The subset of blending features described by **Blend**modesEquation, BlendColor, and the **BlendFunc** ONE_MINUS_CONSTANT_COLOR, CONSTANT_ALPHA, and CONSTANT_COLOR, ONE_MINUS_CONSTANT_ALPHA. These are described separately in section 4.1.6.

The imaging subset is supported only if the EXTENSIONS string includes the substring "ARB_imaging". Querying EXTENSIONS is described in section 6.1.11.

If the imaging subset is not supported, the related pixel transfer operations are not performed; pixels are passed unchanged to the next operation.

Parameter Name	Type	Initial Value	Valid Range
MAP_COLOR	boolean	FALSE	TRUE/FALSE
MAP_STENCIL	boolean	FALSE	TRUE/FALSE
INDEX_SHIFT	integer	0	$(-\infty,\infty)$
INDEX_OFFSET	integer	0	$(-\infty,\infty)$
x_SCALE	float	1.0	$(-\infty,\infty)$
DEPTH_SCALE	float	1.0	$(-\infty,\infty)$
x_BIAS	float	0.0	$(-\infty,\infty)$
DEPTH_BIAS	float	0.0	$(-\infty,\infty)$
POST_CONVOLUTION_x_SCALE	float	1.0	$(-\infty,\infty)$
POST_CONVOLUTION_x_BIAS	float	0.0	$(-\infty,\infty)$
POST_COLOR_MATRIX_x_SCALE	float	1.0	$(-\infty,\infty)$
POST_COLOR_MATRIX_X_BIAS	float	0.0	$(-\infty,\infty)$

Table 3.2: **PixelTransfer** parameters. x is RED, GREEN, BLUE, or ALPHA.

3.6.3 Pixel Transfer Modes

Pixel transfer modes affect the operation of **DrawPixels** (section 3.6.4), **ReadPixels** (section 4.3.2), and **CopyPixels** (section 4.3.3) at the time when one of these commands is executed (which may differ from the time the command is issued). Some pixel transfer modes are set with

```
void PixelTransfer{if}( enum param, T value );
```

param is a symbolic constant indicating a parameter to be set, and value is the value to set it to. Table 3.2 summarizes the pixel transfer parameters that are set with **PixelTransfer**, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error INVALID_VALUE. The same versions of the command exist as for **PixelStore**, and the same rules apply to accepting and converting passed values to set parameters.

The pixel map lookup tables are set with

```
void PixelMap{ui us f}v( enum map, sizei size, T values );
```

map is a symbolic map name, indicating the map to set, *size* indicates the size of the map, and *values* is a pointer to an array of *size* map values.

Map Name	Address	Value	Init. Size	Init. Value
PIXEL_MAP_I_TO_I	color idx	color idx	1	0.0
PIXEL_MAP_S_TO_S	stencil idx	stencil idx	1	0
PIXEL_MAP_I_TO_R	color idx	R	1	0.0
PIXEL_MAP_I_TO_G	color idx	G	1	0.0
PIXEL_MAP_I_TO_B	color idx	В	1	0.0
PIXEL_MAP_I_TO_A	color idx	Α	1	0.0
PIXEL_MAP_R_TO_R	R	R	1	0.0
PIXEL_MAP_G_TO_G	G	G	1	0.0
PIXEL_MAP_B_TO_B	В	В	1	0.0
PIXEL_MAP_A_TO_A	A	A	1	0.0

Table 3.3: **PixelMap** parameters.

The entries of a table may be specified using one of three types: singleprecision floating-point, unsigned short integer, or unsigned integer, depending on which of the three versions of **PixelMap** is called. A table entry is converted to the appropriate type when it is specified. An entry giving a color component value is converted according to table 2.6. An entry giving a color index value is converted from an unsigned short integer or unsigned integer to floating-point. An entry giving a stencil index is converted from single-precision floating-point to an integer by rounding to nearest. The various tables and their initial sizes and entries are summarized in table 3.3. A table that takes an index as an address must have $size = 2^n$ or the error **INVALID_VALUE** results. The maximum allowable *size* of each table is specified by the implementation dependent value MAX_PIXEL_MAP_TABLE, but must be at least 32 (a single maximum applies to all tables). The error **INVALID_VALUE** is generated if a *size* larger than the implemented maximum, or less than one, is given to **PixelMap**.

Color Table Specification

Color lookup tables are specified with

void ColorTable(enum target, enum internalformat, sizei width, enum format, enum type, void *data);

target must be one of the regular color table names listed in table 3.4 to define the table. A proxy table name is a special case discussed later in

Table Name	Type
COLOR_TABLE	regular
POST_CONVOLUTION_COLOR_TABLE	
POST_COLOR_MATRIX_COLOR_TABLE	
PROXY_COLOR_TABLE	proxy
PROXY_POST_CONVOLUTION_COLOR_TABLE	
PROXY_POST_COLOR_MATRIX_COLOR_TABLE	

Table 3.4: Color table names. Regular tables have associated image data. Proxy tables have no image data, and are used only to determine if an image can be loaded into the corresponding regular table.

this section. width, format, type, and data specify an image in memory with the same meaning and allowed values as the corresponding arguments to **DrawPixels** (see section 3.6.4), with *height* taken to be 1. The maximum allowable width of a table is implementation-dependent, but must be at least 32. The formats COLOR_INDEX, DEPTH_COMPONENT, and STENCIL_INDEX and the type BITMAP are not allowed.

The specified image is taken from memory and processed just as if **DrawPixels** were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four COLOR_TABLE_SCALE parameters, biased by the four COLOR_TABLE_BIAS parameters, and clamped to [0, 1]. These parameters are set by calling ColorTableParameterfv as described below.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the *base internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 3.8.1). *internalformat* must be one of the formats in table 3.15 or table 3.16.

The color lookup table is redefined to have width entries, each with the specified internal format. The table is formed with indices 0 through width - 1. Table location *i* is specified by the *i*th image pixel, counting from zero.

The error INVALID_VALUE is generated if *width* is not zero or a non-negative power of two. The error TABLE_TOO_LARGE is generated if the specified color lookup table is too large for the implementation.

The scale and bias parameters for a table are specified by calling

void ColorTableParameter{if}v(enum target, enum pname, T params); *target* must be a regular color table name. *pname* is one of COLOR_TABLE_SCALE or COLOR_TABLE_BIAS. *params* points to an array of four values: red, green, blue, and alpha, in that order.

A GL implementation may vary its allocation of internal component resolution based on any **ColorTable** parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. Allocations must be invariant; the same allocation must be made each time a color table is specified with the same parameter values. These allocation rules also apply to proxy color tables, which are described later in this section.

Alternate Color Table Specification Commands

Color tables may also be specified using image data taken directly from the framebuffer, and portions of existing tables may be respecified.

The command

defines a color table in exactly the manner of **ColorTable**, except that table data are taken from the framebuffer, rather than from client memory. *target* must be a regular color table name. x, y, and width correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image's width and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to **COLOR** and *height* set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for **ColorTable**, beginning with scaling by COLOR_TABLE_SCALE. Parameters *target*, *internalformat* and *width* are specified using the same values, with the same meanings, as the equivalent arguments of **ColorTable**. *format* is taken to be RGBA.

Two additional commands,

```
void ColorSubTable( enum target, sizei start,
    sizei count, enum format, enum type, void *data );
void CopyColorSubTable( enum target, sizei start,
    int x, int y, sizei count );
```

respecify only a portion of an existing color table. No change is made to the *internalformat* or *width* parameters of the specified color table, nor is any

change made to table entries outside the specified portion. *target* must be a regular color table name.

ColorSubTable arguments format, type, and data match the corresponding arguments to **ColorTable**, meaning that they are specified using the same values, and have the same meanings. Likewise, **CopyColorSubTable** arguments x, y, and count match the x, y, and width arguments of **CopyColorTable**. Both of the **ColorSubTable** commands interpret and process pixel groups in exactly the manner of their **ColorTable** counterparts, except that the assignment of R, G, B, and A pixel group values to the color table components is controlled by the internalformat of the table, not by an argument to the command.

Arguments *start* and *count* of **ColorSubTable** and **CopyColorSub-Table** specify a subregion of the color table starting at index *start* and ending at index *start* + *count* - 1. Counting from zero, the *n*th pixel group is assigned to the table entry with index *count* + *n*. The error INVALID_VALUE is generated if *start* + *count* > *width*.

Color Table State and Proxy State

The state necessary for color tables can be divided into two categories. For each of the three tables, there is an array of values. Each array has associated with it a width, an integer describing the internal format of the table, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the table, and two groups of four floating-point numbers to store the table scale and bias. Each initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the scale parameters is (1,1,1,1) and the initial value of the bias parameters is (0,0,0,0).

In addition to the color lookup tables, partially instantiated proxy color lookup tables are maintained. Each proxy table includes width and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy tables do not include image data, nor do they include scale and bias parameters. When **ColorTable** is executed with *target* specified as one of the proxy color table names listed in table 3.4, the proxy state values of the table are recomputed and updated. If the table is too large, no error is generated, but the proxy format, width and component resolutions are set to zero. If the color table would be accommodated by **ColorTable** called with *target* set to the corresponding regular table name (COLOR_TABLE is the regular name corresponding to PROXY_COLOR_TABLE, for example), the proxy state values are set exactly as though the regular table were being specified. Calling **ColorTable** with a proxy *target* has no effect on the image or state of any actual color table.

There is no image associated with any of the proxy targets. They cannot be used as color tables, and they must never be queried using **GetColorTable**. The error INVALID_ENUM is generated if this is attempted.

Convolution Filter Specification

A two-dimensional convolution filter image is specified by calling

void ConvolutionFilter2D(enum target, enum internalformat, sizei width, sizei height, enum format, enum type, void *data);

target must be CONVOLUTION_2D. width, height, format, type, and data specify an image in memory with the same meaning and allowed values as the corresponding parameters to **DrawPixels**. The formats COLOR_INDEX, DEPTH_COMPONENT, and STENCIL_INDEX and the type BITMAP are not allowed.

The specified image is extracted from memory and processed just as if **DrawPixels** were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four two-dimensional CONVOLUTION_FILTER_SCALE parameters and biased by the four two-dimensional CONVOLUTION_FILTER_BIAS parameters. These parameters are set by calling **ConvolutionParameterfv** as described below. No clamping takes place at any time during this process.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the *base internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 3.8.1). *internalformat* must be one of the formats in table 3.15 or table 3.16.

The red, green, blue, alpha, luminance, and/or intensity components of the pixels are stored in floating point, rather than integer format. They form a two-dimensional image indexed with coordinates i, j such that i increases from left to right, starting at zero, and j increases from bottom to top, also starting at zero. Image location i, j is specified by the Nth pixel, counting from zero, where

$$N = i + j * width$$

The error INVALID_VALUE is generated if width or height is greater than the maximum supported value. These values are queried with **GetCon**volutionParameteriv, setting target to CONVOLUTION_2D and pname to MAX_CONVOLUTION_WIDTH or MAX_CONVOLUTION_HEIGHT, respectively. The scale and bias parameters for a two-dimensional filter are specified by calling

```
void ConvolutionParameter{if}v( enum target,
    enum pname, T params );
```

with *target* CONVOLUTION_2D. *pname* is one of CONVOLUTION_FILTER_SCALE or CONVOLUTION_FILTER_BIAS. *params* points to an array of four values: red, green, blue, and alpha, in that order.

A one-dimensional convolution filter is defined using

```
void ConvolutionFilter1D( enum target,
    enum internalformat, sizei width, enum format,
    enum type, void *data );
```

target must be CONVOLUTION_1D. internalformat, width, format, and type have identical semantics and accept the same values as do their two-dimensional counterparts. data must point to a one-dimensional image, however.

The image is extracted from memory and processed as if **Con-volutionFilter2D** were called with a *height* of 1, except that it is scaled and biased by the one-dimensional CONVOLUTION_FILTER_SCALE and CONVOLUTION_FILTER_BIAS parameters. These parameters are specified exactly as the two-dimensional parameters, except that **ConvolutionParameters** is called with *target* CONVOLUTION_1D.

The image is formed with coordinates i such that i increases from left to right, starting at zero. Image location i is specified by the *i*th pixel, counting from zero.

The error INVALID_VALUE is generated if *width* is greater than the maximum supported value. This value is queried using **GetConvolutionParameteriv**, setting *target* to CONVOLUTION_1D and *pname* to MAX_CONVOLUTION_WIDTH.

Special facilities are provided for the definition of two-dimensional *separable* filters – filters whose image can be represented as the product of two one-dimensional images, rather than as full two-dimensional images. A two-dimensional separable convolution filter is specified with

```
void SeparableFilter2D( enum target, enum internalformat,
    sizei width, sizei height, enum format, enum type,
    void *row, void *column );
```

target must be SEPARABLE_2D. internalformat specifies the formats of the table entries of the two one-dimensional images that will be retained. row points to a width pixel wide image of the specified format and type. column points to a height pixel high image, also of the specified format and type.

The two images are extracted from memory and processed as if **ConvolutionFilter1D** were called separately for each, except that each image is scaled and biased by the two-dimensional separable **CONVOLUTION_FILTER_SCALE** and **CONVOLUTION_FILTER_BIAS** parameters. These parameters are specified exactly as the one-dimensional and two-dimensional parameters, except that **ConvolutionParameteriv** is called with *target* **SEPARABLE_2D**.

Alternate Convolution Filter Specification Commands

One and two-dimensional filters may also be specified using image data taken directly from the framebuffer.

The command

```
void CopyConvolutionFilter2D( enum target,
    enum internalformat, int x, int y, sizei width,
    sizei height);
```

defines a two-dimensional filter in exactly the manner of **ConvolutionFilter2D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be CONVOLUTION_2D. x, y, width, and height correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image's width and height, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyP-ixels** with argument *type* set to COLOR, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for Convolution-Filter2D, beginning with scaling by CONVOLUTION_FILTER_SCALE. Parameters *target, internalformat, width,* and *height* are specified using the same values, with the same meanings, as the equivalent arguments of ConvolutionFilter2D. *format* is taken to be RGBA.

The command

```
void CopyConvolutionFilter1D( enum target,
    enum internalformat, int x, int y, sizei width );
```

defines a one-dimensional filter in exactly the manner of **ConvolutionFilter1D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be CONVOLUTION_1D. x, y, and width correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image's width and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to COLOR and *height* set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for Convolution-Filter1D, beginning with scaling by CONVOLUTION_FILTER_SCALE. Parameters *target*, *internalformat*, and *width* are specified using the same values, with the same meanings, as the equivalent arguments of ConvolutionFilter2D. *format* is taken to be RGBA.

Convolution Filter State

The required state for convolution filters includes a one-dimensional image array, two one-dimensional image arrays for the separable filter, and a twodimensional image array. The two-dimensional array has associated with it a height. Each array has associated with it a width, an integer describing the internal format of the table, and six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the table. Each filter (one-dimensional, two-dimensional, and two-dimensional separable) also has associated with it two groups of four floating-point numbers to store the filter scale and bias.

Each initial convolution filter is null (zero width and height, internal format RGBA, with zero-sized components). The initial value of all scale parameters is (1,1,1,1) and the initial value of all bias parameters is (0,0,0,0).

Color Matrix Specification

Setting the matrix mode to COLOR causes the matrix operations described in section 2.10.2 to apply to the top matrix on the color matrix stack. All matrix operations have the same effect on the color matrix as they do on the other matrices.

Histogram Table Specification

The histogram table is specified with

3.6. PIXEL RECTANGLES

void Histogram(enum target, sizei width, enum internalformat, boolean sink);

target must be HISTOGRAM if a histogram table is to be specified. target value PROXY_HISTOGRAM is a special case discussed later in this section. width specifies the number of entries in the histogram table, and internalformat specifies the format of each table entry. The maximum allowable width of the histogram table is implementation-dependent, but must be at least 32. sink specifies whether pixel groups will be consumed by the histogram operation (TRUE) or passed on to the minmax operation (FALSE).

If no error results from the execution of **Histogram**, the specified histogram table is redefined to have *width* entries, each with the specified internal format. The entries are indexed 0 through *width* -1. Each component in each entry is set to zero. The values in the previous histogram table, if any, are lost.

The error INVALID_VALUE is generated if *width* is not zero or a non-negative power of 2. The error TABLE_TOO_LARGE is generated if the specified histogram table is too large for the implementation. The error INVALID_ENUM is generated if *internalformat* is not one of the values accepted by the corresponding parameter of TexImage2D, or is 1, 2, 3, 4, INTENSITY, INTENSITY4, INTENSITY8, INTENSITY12, or INTENSITY16.

A GL implementation may vary its allocation of internal component resolution based on any **Histogram** parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. In particular, allocations must be invariant; the same allocation must be made each time a histogram is specified with the same parameter values. These allocation rules also apply to the proxy histogram, which is described later in this section.

Histogram State and Proxy State

The state necessary for histogram operation is an array of values, with which is associated a width, an integer describing the internal format of the histogram, five integer values describing the resolutions of each of the red, green, blue, alpha, and luminance components of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the flag is false.

In addition to the histogram table, a partially instantiated proxy histogram table is maintained. It includes width, internal format, and red, green, blue, alpha, and luminance component resolutions. The proxy table does not include image data or the flag. When **Histogram** is executed with *target* set to **PROXY_HISTOGRAM**, the proxy state values are recomputed and updated. If the histogram array is too large, no error is generated, but the proxy format, width, and component resolutions are set to zero. If the histogram table would be accomodated by **Histogram** called with *target* set to **HISTOGRAM**, the proxy state values are set exactly as though the actual histogram table were being specified. Calling **Histogram** with *target* **PROXY_HISTOGRAM** has no effect on the actual histogram table.

There is no image associated with PROXY_HISTOGRAM. It cannot be used as a histogram, and its image must never queried using **GetHistogram**. The error INVALID_ENUM results if this is attempted.

Minmax Table Specification

The minmax table is specified with

target must be MINMAX. internal format specifies the format of the table entries. sink specifies whether pixel groups will be consumed by the minmax operation (TRUE) or passed on to final conversion (FALSE).

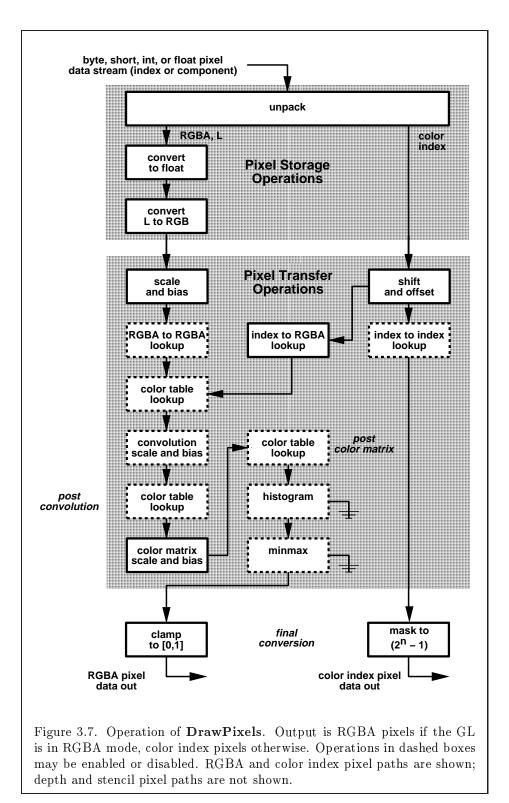
The error INVALID_ENUM is generated if *internalformat* is not one of the values accepted by the corresponding parameter of **TexImage2D**, or is 1, 2, 3, 4, INTENSITY, INTENSITY4, INTENSITY8, INTENSITY12, or INTENSITY16. The resulting table always has 2 entries, each with values corresponding only to the components of the internal format.

The state necessary for minmax operation is a table containing two elements (the first element stores the minimum values, the second stores the maximum values), an integer describing the internal format of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial state is a minimum table entry set to the maximum representable value and a maximum table entry set to the minimum representable value. Internal format is set to RGBA and the initial value of the flag is false.

3.6.4 Rasterization of Pixel Rectangles

The process of drawing pixels encoded in host memory is diagrammed in figure 3.7. We describe the stages of this process in the order in which they occur.

3.6. PIXEL RECTANGLES



Pixels are drawn using

void DrawPixels(sizei width, sizei height, enum format, enum type, void *data);

format is a symbolic constant indicating what the values in memory represent. width and height are the width and height, respectively, of the pixel rectangle to be drawn. data is a pointer to the data to be drawn. These data are represented with one of seven GL data types, specified by type. The correspondence between the twenty type token values and the GL data types they indicate is given in table 3.5. If the GL is in color index mode and format is not one of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT, then the error INVALID_OPERATION occurs. If type is BITMAP and format is not COLOR_INDEX or STENCIL_INDEX then the error INVALID_ENUM occurs. Some additional constraints on the combinations of format and type values that are accepted is discussed below.

Unpacking

Data are taken from host memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned short integers (GL data types short and ushort), signed or unsigned integers (GL data types int and uint), or floating point values (GL data type float). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 3.6 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield components.

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If UNPACK_SWAP_BYTES is enabled, however, then the values are interpreted with the bit orderings modified as per table 3.7. The modified bit orderings are defined only if the GL data type ubyte has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by the pointer passed to **DrawPixels**. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is *width*; otherwise the number of groups is UNPACK_ROW_LENGTH. If p indicates the location in memory of the first element of the first row, then the first element of the Nth row is indicated by

<i>type</i> Parameter	Corresponding	Special
Token Name	GL Data Type	Interpretation
UNSIGNED_BYTE	ubyte	No
BITMAP	ubyte	Yes
BYTE	byte	No
UNSIGNED_SHORT	ushort	No
SHORT	short	No
UNSIGNED_INT	uint	No
INT	int	No
FLOAT	float	No
UNSIGNED_BYTE_3_3_2	ubyte	Yes
UNSIGNED_BYTE_2_3_3_REV	ubyte	Yes
UNSIGNED_SHORT_5_6_5	ushort	Yes
UNSIGNED_SHORT_5_6_5_REV	ushort	Yes
UNSIGNED_SHORT_4_4_4_4	ushort	Yes
UNSIGNED_SHORT_4_4_4_REV	ushort	Yes
UNSIGNED_SHORT_5_5_5_1	ushort	Yes
UNSIGNED_SHORT_1_5_5_5_REV	ushort	Yes
UNSIGNED_INT_8_8_8_8	uint	Yes
UNSIGNED_INT_8_8_8_8_REV	uint	Yes
UNSIGNED_INT_10_10_10_2	uint	Yes
UNSIGNED_INT_2_10_10_REV	uint	Yes

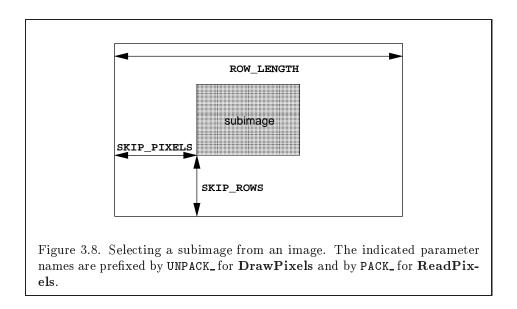
Table 3.5: **DrawPixels** and **ReadPixels** *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6.4.

Format Name	Element Meaning and Order	Target Buffer
COLOR_INDEX	Color Index	Color
STENCIL_INDEX	Stencil Index	Stencil
DEPTH_COMPONENT	${ m Depth}$	Depth
RED	R	Color
GREEN	G	Color
BLUE	В	Color
ALPHA	А	Color
RGB	R, G, B	Color
RGBA	R, G, B, A	Color
BGR	B, G, R	Color
BGRA	$\mathbf{B},\ \mathbf{G},\ \mathbf{R},\ \mathbf{A}$	Color
LUMINANCE	Luminance	Color
LUMINANCE_ALPHA	Luminance, A	Color

Table 3.6: **DrawPixels** and **ReadPixels** formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components.

Element Size	Default Bit Ordering	Modified Bit Ordering
8 bit	[70]	[70]
16 bit	[150]	[70][158]
32 bit	[310]	[70][158][2316][3124]

Table 3.7: Bit ordering modification of elements when UNPACK_SWAP_BYTES is enabled. These reorderings are defined only when GL data type ubyte has 8 bits, and then only for GL data types with 8, 16, or 32 bits. Bit 0 is the least significant.



$$p + Nk \tag{3.8}$$

where N is the row number (counting from zero) and k is defined as

$$k = \begin{cases} nl & s \ge a, \\ a/s \lceil snl/a \rceil & s < a \end{cases}$$
(3.9)

where n is the number of elements in a group, l is the number of groups in the row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubytes, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then k = nl for all values of a.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the pointer supplied to **DrawPixels** is effectively advanced by (UNPACK_SKIP_PIXELS) $n + (\text{UNPACK}_SKIP_ROWS)k$ elements. Then width groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. height sets of width groups of values are obtained this way. See figure 3.8.

Calling **DrawPixels** with a *type* of UNSIGNED_BYTE_3_3_2, UNSIGNED_BYTE_2_3_3_REV, UNSIGNED_SHORT_5_6_5, UNSIGNED_SHORT_5_6_5_REV,

type Parameter	GL Data	Number of	Matching
Token Name	Type	Components	Pixel Formats
UNSIGNED_BYTE_3_3_2	ubyte	3	RGB
UNSIGNED_BYTE_2_3_3_REV	ubyte	3	RGB
UNSIGNED_SHORT_5_6_5	ushort	3	RGB
UNSIGNED_SHORT_5_6_5_REV	ushort	3	RGB
UNSIGNED_SHORT_4_4_4_4	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_4_4_4_REV	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_5_5_5_1	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_1_5_5_REV	ushort	4	RGBA,BGRA
UNSIGNED_INT_8_8_8_8	uint	4	RGBA,BGRA
UNSIGNED_INT_8_8_8_8_REV	uint	4	RGBA,BGRA
UNSIGNED_INT_10_10_10_2	uint	4	RGBA,BGRA
UNSIGNED_INT_2_10_10_REV	uint	4	RGBA,BGRA

Table 3.8: Packed pixel formats.

UNSIGNED_SHORT_4_4_4_4, UNSIGNED_SHORT_4_4_4_REV, UNSIGNED_SHORT_5_5_5_1, UNSIGNED_SHORT_1_5_5_5_REV, UNSIGNED_INT_8_8_8_8, UNSIGNED_INT_8_8_8_8_REV, UNSIGNED_INT_10_10_10_2, or UNSIGNED_INT_2_10_10_10_REV is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.8. The error INVALID_OPERATION is generated if a mismatch occurs. This constraint also holds for all other functions that accept or return pixel data using *type* and *format* parameters to define the type and format of that data.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.9, 3.10, and 3.11. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are right-justified in the pixel.

Components are normally packed with the first component in the most significant bits of the bitfield, and successive component occupying progressively less significant locations. Types whose token names end with _REV reverse the component packing order from least to most significant locations. In all cases, the most significant bit of each component is packed in

3.6. PIXEL RECTANGLES

the most significant bit location of its location in the bitfield. UNSIGNED_BYTE_3_3_2:



UNSIGNED_BYTE_2_3_3_REV:



Table 3.9: UNSIGNED_BYTE formats. Bit numbers are indicated for each component.

UNSIGNED_SHORT_5_6_5:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1st	Compo	nent				2	nd					3rd		

UNSIGNED_SHORT_5_6_5_REV:

15	14	13	12	11 10 9 8 7						5	4	3	2	1	0
		3rd					2	2nd				1s	t Compo	nent	

UNSIGNED_SHORT_4_4_4_4:

18	5	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	lst Com	ponent			21	nd				3 rd				4th	

UNSIGNED_SHORT_4_4_4_REV:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		4th			3	rd				2nd			1st Co	mponen	t

UNSIGNED_SHORT_5_5_1:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component 2nd												3rd			4th

UNSIGNED_SHORT_1_5_5_5_REV:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
4th			3rd					2nd				1s	t Compo	onent	

Table 3.10: UNSIGNED_SHORT formats

UNSIGNED_INT_8_8_8_8:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1st Component 2						2r	nd							3r	d							41	th							

UNSIGNED_INT_8_8_8_REV:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			4t	h							3r	d							2r	nd						1st	Cor	npo	nent		

UNSIGNED_INT_10_10_10_2:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			1st	Corr	npon	nent								2r	nd									3	rd					41	th

UNSIGNED_INT_2_10_10_REV:

31 30	29 28	3 27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
4th				3 r	d									2n	d								1st	Cor	npor	nent			

Table 3.11: UNSIGNED_INT formats

Format	First	Second	Third	Fourth
	Component	Component	Component	Component
RGB	red	green	blue	
RGBA	red	green	blue	alpha
BGRA	blue	green	red	alpha

Table 3.12: Packed pixel field assignments

The assignment of component to fields in the packed pixel is as described in table 3.12

Byte swapping, if enabled, is performed before the component are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if "group" is substituted for "component" and the number of components per group is understood to be one.

Calling **DrawPixels** with a *type* of **BITMAP** is a special case in which the data are a series of GL ubyte values. Each ubyte value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of **UNPACK_LSB_FIRST** is **FALSE**; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each ubyte are not significant.

The first element of the first row is the first bit (as defined above) of the ubyte pointed to by the pointer passed to **DrawPixels**. The first element of the second row is the first bit (again as defined above) of the ubyte at location p + k, where k is computed as

$$k = a \left\lceil \frac{l}{8a} \right\rceil \tag{3.10}$$

There is a mechanism for selecting a sub-rectangle of elements from a BITMAP image as well. Before obtaining the first element from memory, the pointer supplied to **DrawPixels** is effectively advanced by UNPACK_SKIP_ROWS * k ubytes. Then UNPACK_SKIP_PIXELS 1-bit elements are ignored, and the subsequent width 1-bit elements are obtained, without advancing the ubyte pointer, after which the pointer is advanced by k ubytes. height sets of width elements are obtained this way.

Conversion to floating-point

This step applies only to groups of components. It is not performed on indices. Each element in a group is converted to a floating-point value according to the appropriate formula in table 2.6 (section 2.13). For packed pixel types, each element in the group is converted by computing $c / (2^N - 1)$, where c is the unsigned integer value of the bitfield containing the element and N is the number of bits in the bitfield.

Conversion to RGB

This step is applied only if the *format* is LUMINANCE or LUMINANCE_ALPHA. If the *format* is LUMINANCE, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is LUMINANCE_ALPHA, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1.0. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0.0.

Pixel Transfer Operations

This step is actually a sequence of steps. Because the pixel transfer operations are performed equivalently during the drawing, copying, and reading of pixels, and during the specification of texture images (either from memory or from the framebuffer), they are described separately in section 3.6.5. After the processing described in that section is completed, groups are processed as described in the following sections.

Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by $2^n - 1$, where n is the number of bits in an index buffer. For RGBA components, each element is clamped to [0, 1]. The

resulting values are converted to fixed-point according to the rules given in section 2.13.9 (Final Color Processing).

For a depth component, an element is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.10.1, Controlling the Viewport).

Stencil indices are masked by $2^n - 1$, where n is the number of bits in the stencil buffer.

Conversion to Fragments

The conversion of a group to fragments is controlled with

```
void PixelZoom(float z_x, float z_y);
```

Let (x_{rp}, y_{rp}) be the current raster position (section 2.12). (If the current raster position is invalid, then **DrawPixels** is ignored; pixel transfer operations do not update the histogram or minmax tables, and no fragments are generated. However, the histogram and minmax tables are updated even if the corresponding fragments are later rejected by the pixel ownership (section 4.1.1) or scissor (section 4.1.2) tests.) If a particular group (index or components) is the *n*th in a row and belongs to the *m*th row, consider the region in window coordinates bounded by the rectangle with corners

$$(x_{rp} + z_x n, y_{rp} + z_y m)$$
 and $(x_{rp} + z_x (n+1), y_{rp} + z_y (m+1))$

(either z_x or z_y may be negative). Any fragments whose centers lie inside of this rectangle (or on its bottom or left boundaries) are produced in correspondence with this particular group of elements.

A fragment arising from a group consisting of color data takes on the color index or color components of the group; the depth and texture coordinates are taken from the current raster position's associated data. A fragment arising from a depth component takes the component's depth value; the color and texture coordinates are given by those associated with the current raster position. In both cases texture coordinates s, t, and r are replaced with s/q, t/q, and r/q, respectively. If q is less than or equal to zero, the results are undefined. Groups arising from **DrawPixels** with a *format* of **STENCIL_INDEX** are treated specially and are described in section 4.3.1.

3.6.5 Pixel Transfer Operations

The GL defines four kinds of pixel groups:

100

- 1. *RGBA component:* Each group comprises four color components: red, green, blue, and alpha.
- 2. Depth component: Each group comprises a single depth component.
- 3. Color index: Each group comprises a single color index.
- 4. Stencil index: Each group comprises a single stencil index.

Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if an operation is not applicable to a given group, it is skipped.

Arithmetic on Components

This step applies only to RGBA component and depth component groups. Each component is multiplied by an appropriate signed scale factor: RED_SCALE for an R component, GREEN_SCALE for a G component, BLUE_SCALE for a B component, and ALPHA_SCALE for an A component, or DEPTH_SCALE for a depth component. Then the result is added to the appropriate signed bias: RED_BIAS, GREEN_BIAS, BLUE_BIAS, ALPHA_BIAS, or DEPTH_BIAS.

Arithmetic on Indices

This step applies only to color index and stencil index groups. If the index is a floating-point value, it is converted to fixed-point, with an unspecified number of bits to the right of the binary point and at least $\lceil \log_2(MAX_PIXEL_MAP_TABLE) \rceil$ bits to the left of the binary point. Indices that are already integers remain so; any fraction bits in the resulting fixed-point value are zero.

The fixed-point index is then shifted by $|INDEX_SHIFT|$ bits, left if $INDEX_SHIFT > 0$ and right otherwise. In either case the shift is zero-filled. Then, the signed integer offset INDEX_OFFSET is added to the index.

RGBA to **RGBA** Lookup

This step applies only to RGBA component groups, and is skipped if MAP_COLOR is FALSE. First, each component is clamped to the range [0, 1]. There is a table associated with each of the R, G, B, and A component elements: PIXEL_MAP_R_TO_R for R, PIXEL_MAP_G_TO_G for G, PIXEL_MAP_B_TO_B for B, and PIXEL_MAP_A_TO_A for A. Each element is multiplied by an integer one less than the size of the corresponding table, and, for each element, an

address is found by rounding this value to the nearest integer. For each element, the addressed value in the corresponding table replaces the element.

Color Index Lookup

This step applies only to color index groups. If the GL command that invokes the pixel transfer operation requires that RGBA component pixel groups be generated, then a conversion is performed at this step. RGBA component pixel groups are required if

- 1. The groups will be rasterized, and the GL is in RGBA mode, or
- 2. The groups will be loaded as an image into texture memory, or
- 3. The groups will be returned to client memory with a format other than COLOR_INDEX.

If RGBA component groups are required, then the integer part of the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A. Each of these tables must have 2^n entries for some integer value of n (n may be different for each table). For each table, the index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The indexed value becomes an R, G, B, or A value, as appropriate. The group of four elements so obtained replaces the index, changing the group's type to RGBA component.

If RGBA component groups are not required, and if MAP_COLOR is enabled, then the index is looked up in the PIXEL_MAP_I_TO_I table (otherwise, the index is not looked up). Again, the table must have 2^n entries for some integer n. The index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The value in the table replaces the index. The floating-point table value is first rounded to a fixed-point value with unspecified precision. The group's type remains color index.

Stencil Index Lookup

This step applies only to stencil index groups. If MAP_STENCIL is enabled, then the index is looked up in the PIXEL_MAP_S_TO_S table (otherwise, the index is not looked up). The table must have 2^n entries for some integer n. The integer index is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The integer value in the table replaces the index.

Base Internal Format	R	G	В	Α
ALPHA				A_t
LUMINANCE	L_t	L_t	L_t	
LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
INTENSITY	I_t	I_t	I_t	I_t
RGB	R_t	G_t	B_t	
RGBA	R_t	G_t	B_t	A_t

Table 3.13: Color table lookup. R_t , G_t , B_t , A_t , L_t , and I_t are color table values that are assigned to pixel components R, G, B, and A depending on the table format. When there is no assignment, the component value is left unchanged by lookup.

Color Table Lookup

This step applies only to RGBA component groups. Color table lookup is only done if COLOR_TABLE is enabled. If a zero-width table is enabled, no lookup is performed.

The internal format of the table determines which components of the group will be replaced (see table 3.13). The components to be replaced are converted to indices by clamping to [0, 1], multiplying by an integer one less than the width of the table, and rounding to the nearest integer. Components are replaced by the table entry at the index.

The required state is one bit indicating whether color table lookup is enabled or disabled. In the initial state, lookup is disabled.

Convolution

This step applies only to RGBA component groups. If CONVOLUTION_1D is enabled, the one-dimensional convolution filter is applied only to the onedimensional texture images passed to TexImage1D, TexSubImage1D, CopyTexImage1D, and CopyTexSubImage1D, and returned by Get-TexImage (see section 6.1.4) with target TEXTURE_1D. If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to DrawPixels, CopyPixels, ReadPixels, TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTex-SubImage2D, and CopyTexSubImage3D, and returned by GetTexImage with target TEXTURE_2D. If SEPARABLE_2D is enabled, and CONVOLUTION_2D is disabled, the separable two-dimensional convolution filter is instead ap-

Base Filter Format	R	G	В	А
ALPHA	R_s	G_s	B_s	$A_s * A_f$
LUMINANCE	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	A_s
LUMINANCE_ALPHA	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	$A_s * A_f$
INTENSITY	$R_s * I_f$	$G_s * I_f$	$B_s * I_f$	$A_s * I_f$
RGB	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	A_s
RGBA	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	$A_s * A_f$

Table 3.14: Computation of filtered color components depending on filter image format. C * F indicates the convolution of image component C with filter F.

plied these images.

The convolution operation is a sum of products of source image pixels and convolution filter pixels. Source image pixels always have four components: red, green, blue, and alpha, denoted in the equations below as R_s , G_s , B_s , and A_s . Filter pixels may be stored in one of five formats, with 1, 2, 3, or 4 components. These components are denoted as R_f , G_f , B_f , A_f , L_f , and I_f in the equations below. The result of the convolution operation is the 4-tuple R,G,B,A. Depending on the internal format of the filter, individual color components of each source image pixel are convolved with one filter component, or are passed unmodified. The rules for this are defined in table 3.14.

The convolution operation is defined differently for each of the three convolution filters. The variables W_f and H_f refer to the dimensions of the convolution filter. The variables W_s and H_s refer to the dimensions of the source pixel image.

The convolution equations are defined as follows, where C refers to the filtered result, C_f refers to the one- or two-dimensional convolution filter, and C_{row} and C_{column} refer to the two one-dimensional filters comprising the two-dimensional separable filter. C'_s depends on the source image color C_s and the convolution border mode as described below. C_r , the filtered output image, depends on all of these variables and is described separately for each border mode. The pixel indexing nomenclature is decribed in the **Convolution Filter Specification** subsection of section 3.6.3.

One-dimensional filter:

$$C[i'] = \sum_{n=0}^{W_f - 1} C'_s[i' + n] * C_f[n]$$

Two-dimensional filter:

$$C[i',j'] = \sum_{n=0}^{W_f - 1} \sum_{m=0}^{H_f - 1} C'_s[i' + n, j' + m] * C_f[n,m]$$

Two-dimensional separable filter:

$$C[i',j'] = \sum_{n=0}^{W_f - 1} \sum_{m=0}^{H_f - 1} C'_s[i' + n, j' + m] * C_{row}[n] * C_{column}[m]$$

If W_f of a one-dimensional filter is zero, then C[i] is always set to zero. Likewise, if either W_f or H_f of a two-dimensional filter is zero, then C[i, j] is always set to zero.

The convolution border mode for a specific convolution filter is specified by calling

where *target* is the name of the filter, *pname* is CONVOLUTION_BORDER_MODE, and *param* is one of REDUCE, CONSTANT_BORDER or REPLICATE_BORDER.

Border Mode REDUCE

The width and height of source images convolved with border mode REDUCE are reduced by $W_f - 1$ and $H_f - 1$, respectively. If this reduction would generate a resulting image with zero or negative width and/or height, the output is simply null, with no error generated. The coordinates of the image that results from a convolution with border mode REDUCE are zero through $W_s - W_f$ in width, and zero through $H_s - H_f$ in height. In cases where errors can result from the specification of invalid image dimensions, it is these resulting dimensions that are tested, not the dimensions of the source image. (A specific example is **TexImage1D** and **TexImage2D**, which specify constraints for image dimensions. Even if **TexImage1D** or **TexImage2D** is called with a null pixel pointer, the dimensions of the resulting texture image are those that would result from the convolution of the specified image).

When the border mode is REDUCE, C'_s equals the source image color C_s and C_r equals the filtered result C.

For the remaining border modes, define $C_w = \lfloor W_f/2 \rfloor$ and $C_h = \lfloor H_f/2 \rfloor$. The coordinates (C_w, C_h) define the center of the convolution filter.

Border Mode CONSTANT_BORDER

If the convolution border mode is CONSTANT_BORDER, the output image has the same dimensions as the source image. The result of the convolution is the same as if the source image were surrounded by pixels with the same color as the current convolution border color. Whenever the convolution filter extends beyond one of the edges of the source image, the constant-color border pixels are used as input to the filter. The current convolution border color is set by calling **ConvolutionParameterfv** or **ConvolutionParameteriv** with *pname* set to CONVOLUTION_BORDER_COLOR and *params* containing four values that comprise the RGBA color to be used as the image border. Integer color components are interpreted linearly such that the most positive integer maps to 1.0, and the most negative integer maps to -1.0. Floating point color components are not clamped when they are specified.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where C[i'] is computed using the following equation for $C'_{s}[i']$:

$$C_s'[i'] = \left\{ egin{array}{cc} C_s[i'], & 0 \leq i' < W_s \ C_c, & otherwise \end{array}
ight.$$

and C_c is the convolution border color.

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i,j] = C[i - C_w, j - C_h]$$

where C[i', j'] is computed using the following equation for $C'_s[i', j']$:

$$C_s'[i',j'] = \left\{ egin{array}{cc} C_s[i',j'], & 0 \leq i' < W_s, 0 \leq j' < H_s \ C_c, & otherwise \end{array}
ight.$$

Border Mode REPLICATE_BORDER

The convolution border mode REPLICATE_BORDER also produces an output image with the same dimensions as the source image. The behavior of this mode is identical to that of the CONSTANT_BORDER mode except for the treatment of pixel locations where the convolution filter extends beyond the edge of the source image. For these locations, it is as if the outermost onepixel border of the source image was replicated. Conceptually, each pixel in

106

the leftmost one-pixel column of the source image is replicated C_w times to provide additional image data along the left edge, each pixel in the rightmost one-pixel column is replicated C_w times to provide additional image data along the right edge, and each pixel value in the top and bottom one-pixel rows is replicated to create C_h rows of image data along the top and bottom edges. The pixel value at each corner is also replicated in order to provide data for the convolution operation at each corner of the source image.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where C[i'] is computed using the following equation for $C'_s[i']$:

$$C'_s[i'] = C_s[\operatorname{clamp}(i', W_s)]$$

and the clamping function $\operatorname{clamp}(val, max)$ is defined as

$$\operatorname{clamp}(val, max) = \left\{ egin{array}{cc} 0, & val < 0 \ val, & 0 \leq val < max \ max - 1, & val >= max \end{array}
ight.$$

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i,j] = C[i - C_w, j - C_h]$$

where C[i', j'] is computed using the following equation for $C'_s[i', j']$:

$$C'_{s}[i', j'] = C_{s}[\operatorname{clamp}(i', W_{s}), \operatorname{clamp}(j', H_{s})]$$

After convolution, each component of the resulting image is scaled by the corresponding **PixelTransfer** parameters: **POST_CONVOLUTION_RED_SCALE** for an R component, POST_CONVOLUTION_GREEN_SCALE for a G com-POST_CONVOLUTION_BLUE_SCALE for \mathbf{a} В component, and ponent, POST_CONVOLUTION_ALPHA_SCALE for an Α component. The result added to the corresponding bias: POST_CONVOLUTION_RED_BIAS, is POST_CONVOLUTION_GREEN_BIAS, POST_CONVOLUTION_BLUE_BIAS, or POST_CONVOLUTION_ALPHA_BIAS.

The required state is three bits indicating whether each of onedimensional, two-dimensional, or separable two-dimensional convolution is enabled or disabled, an integer describing the current convolution border mode, and four floating-point values specifying the convolution border color. In the initial state, all convolution operations are disabled, the border mode is **REDUCE**, and the border color is (0, 0, 0, 0).

Post Convolution Color Table Lookup

This step applies only to RGBA component groups. Post convolution color table lookup is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant POST_CONVOLUTION_COLOR_TABLE. The post convolution table is defined by calling **ColorTable** with a *target* argument of POST_CONVOLUTION_COLOR_TABLE. In all other respects, operation is identical to color table lookup, as defined earlier in section 3.6.5.

The required state is one bit indicating whether post convolution table lookup is enabled or disabled. In the initial state, lookup is disabled.

Color Matrix Transformation

This step applies only to RGBA component groups. The components are transformed by the color matrix. Each transformed component is multiplied by an appropriate signed scale factor: POST_COLOR_MATRIX_RED_SCALE for an R component, POST_COLOR_MATRIX_GREEN_SCALE for a G component, POST_COLOR_MATRIX_BLUE_SCALE for a B component, and POST_COLOR_MATRIX_ALPHA_SCALE for an A component. The result is added to a signed bias: POST_COLOR_MATRIX_RED_BIAS, POST_COLOR_MATRIX_GREEN_BIAS, POST_COLOR_MATRIX_BLUE_BIAS, or POST_COLOR_MATRIX_ALPHA_BIAS. The result-ing components replace each component of the original group.

That is, if M_c is the color matrix, a subscript of s represents the scale term for a component, and a subscript of b represents the bias term, then the components

$$\begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

are transformed to

$$\begin{pmatrix} R'\\G'\\B'\\A' \end{pmatrix} = \begin{pmatrix} R_s & 0 & 0 & 0\\ 0 & G_s & 0 & 0\\ 0 & 0 & B_s & 0\\ 0 & 0 & 0 & A_s \end{pmatrix} M_c \begin{pmatrix} R\\G\\B\\A \end{pmatrix} + \begin{pmatrix} R_b\\G_b\\B_b\\A_b \end{pmatrix}.$$

Post Color Matrix Color Table Lookup

This step applies only to RGBA component groups. Post color matrix color table lookup is enabled or disabled by calling **Enable** or **Disable**

108

with the symbolic constant POST_COLOR_MATRIX_COLOR_TABLE. The post color matrix table is defined by calling ColorTable with a *target* argument of POST_COLOR_MATRIX_COLOR_TABLE. In all other respects, operation is identical to color table lookup, as defined in section 3.6.5.

The required state is one bit indicating whether post color matrix lookup is enabled or disabled. In the initial state, lookup is disabled.

Histogram

This step applies only to RGBA component groups. Histogram operation is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant **HISTOGRAM**.

If the width of the table is non-zero, then indices R_i , G_i , B_i , and A_i are derived from the red, green, blue, and alpha components of each pixel group (without modifying these components) by clamping each component to [0, 1], multiplying by one less than the width of the histogram table, and rounding to the nearest integer. If the format of the HISTOGRAM table includes red or luminance, the red or luminance component of histogram entry R_i is incremented by one. If the format of the HISTOGRAM table includes green, the green component of histogram entry G_i is incremented by one. The blue and alpha components of histogram entry G_i is incremented by one. The blue multiply entry R_i and A_i are incremented in the same way. If a histogram entry component is incremented beyond its maximum value, its value becomes undefined; this is not an error.

If the **Histogram** sink parameter is **FALSE**, histogram operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the histogram operation is completed. Because histogram precedes minmax, no minmax operation is performed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

Minmax

This step applies only to RGBA component groups. Minmax operation is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant MINMAX.

If the format of the minmax table includes red or luminance, the red component value replaces the red or luminance value in the minimum table element if and only if it is less than that component. Likewise, if the format includes red or luminance and the red component of the group is greater than the red or luminance value in the maximum element, the red group component replaces the red or luminance maximum component. If the format of the table includes green, the green group component conditionally replaces the green minimum and/or maximum if it is smaller or larger, respectively. The blue and alpha group components are similarly tested and replaced, if the table format includes blue and/or alpha. The internal type of the minimum and maximum component values is floating point, with at least the same representable range as a floating point number used to represent colors (section 2.1.1). There are no semantics defined for the treatment of group component values that are outside the representable range.

If the Minmax *sink* parameter is FALSE, minmax operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the minmax operation is completed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

3.7 Bitmaps

Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. These data are those associated with the *current raster position*.

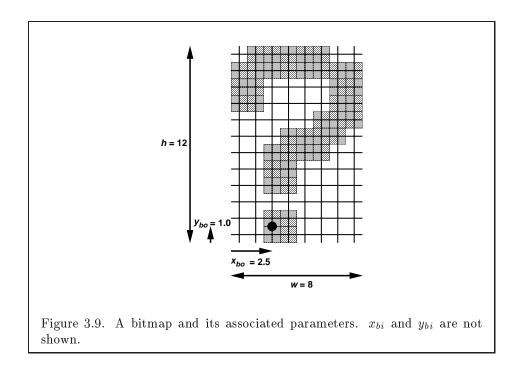
Bitmaps are sent using

```
void \operatorname{Bitmap}( sizei w, sizei h, float x_{bo}, float y_{bo}, float x_{bi}, float y_{bi}, ubyte *data );
```

w and h comprise the integer width and height of the rectangular bitmap, respectively. (x_{bo}, y_{bo}) gives the floating-point x and y values of the bitmap's origin. (x_{bi}, y_{bi}) gives the floating-point x and y increments that are added to the raster position after the bitmap is rasterized. *data* is a pointer to a bitmap.

Like a polygon pattern, a bitmap is unpacked from memory according to the procedure given in section 3.6.4 for **DrawPixels**; it is as if the *width* and *height* passed to that command were equal to w and h, respectively, the *type* were **BITMAP**, and the *format* were **COLOR_INDEX**. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones. See figure 3.9.

A bitmap sent using **Bitmap** is rasterized as follows. First, if the current raster position is invalid (the valid bit is reset), the bitmap is ignored.



Otherwise, a rectangular array of fragments is constructed, with lower left corner at

$$(x_{ll}, y_{ll}) = (\lfloor x_{rp} - x_{bo} \rfloor, \lfloor y_{rp} - y_{bo} \rfloor)$$

and upper right corner at $(x_{ll} + w, y_{ll} + h)$ where w and h are the width and height of the bitmap, respectively. Fragments in the array are produced if the corresponding bit in the bitmap is 1 and not produced otherwise. The associated data for each fragment are those associated with the current raster position, with texture coordinates s, t, and r replaced with s/q, t/q, and r/q, respectively. If q is less than or equal to zero, the results are undefined. Once the fragments have been produced, the current raster position is updated:

$$(x_{rp}, y_{rp}) \leftarrow (x_{rp} + x_{bi}, y_{rp} + y_{bi}).$$

The z and w values of the current raster position remain unchanged.

3.8 Texturing

Texturing maps a portion of a specified image onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a fragment's (s, t, r) coordinates to modify the fragment's primary RGBA color. Texturing does not affect the secondary color.

Texturing is specified only for RGBA mode; its use in color index mode is undefined.

The GL provides a means to specify the details of how texturing of a primitive is effected. These details include specification of the image to be texture mapped, the means by which the image is filtered when applied to the primitive, and the function that determines what RGBA value is produced given a fragment color and an image value.

3.8.1 Texture Image Specification

The command

```
void TexImage3D( enum target, int level,
    int internalformat, sizei width, sizei height,
    sizei depth, int border, enum format, enum type,
    void *data );
```

is used to specify a three-dimensional texture image. *target* must be either **TEXTURE_3D**, or **PROXY_TEXTURE_3D** in the special case discussed in section 3.8.7. *format*, *type*, and *data* match the corresponding arguments to **DrawPixels** (refer to section 3.6.4); they specify the format of the image data, the type of those data, and a pointer to the image data in host memory. The *formats* **STENCIL_INDEX** and **DEPTH_COMPONENT** are not allowed.

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the *width* and *height* parameters to **TexImage3D**. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT control the row-to-row spacing in these images in the same manner as **DrawPixels**. If the value of the integer parameter UNPACK_IMAGE_HEIGHT is not positive, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is UNPACK_IMAGE_HEIGHT. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a three-dimensional image relies on the integer parameter UNPACK_SKIP_IMAGES. If UNPACK_SKIP_IMAGES is positive, the pointer is advanced by UNPACK_SKIP_IMAGES times the number of elements in one two-dimensional image before obtaining the first group from memory. Then *depth* two-dimensional images are processed, each having a subimage extracted in the same manner as **DrawPixels**.

The selected groups are processed exactly as for **DrawPixels**, stopping just before final conversion. Each R, G, B, and A value so generated is clamped to [0, 1].

Components are then selected from the resulting R, G, B, and A values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat*. Table 3.15 summarizes the mapping of R, G, B, and A values to texture components, as a function of the base internal format of the texture image. *internalformat* may be specified as one of the six base internal format symbolic constants listed in table 3.15, or as one of the *sized internal format* symbolic constants listed in table 3.16. *internalformat* may (for backwards compatibility with the 1.0 version of the GL) also take on the integer values 1, 2, 3, and 4, which are equivalent to symbolic constants LUMINANCE, LUMINANCE_ALPHA, RGB, and RGBA respectively. Specifying a value for *internalformat* that is not one of the above values generates the error INVALID_VALUE.

The internal component resolution is the number of bits allocated to each value in a texture image. If internalformat is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15, and the memory allocation per texture component is assigned by the GL to match the allocations listed in table 3.16 as closely as possible. (The definition of closely is left up to the implementation. Implementations are not required to support more than one resolution for each base internal format.)

A GL implementation may vary its allocation of internal component resolution based on any **TexImage3D**, **TexImage2D** (see below), or **TexImage1D** (see below) parameter (except *target*), but the allocation must not be a function of any other state, and cannot be changed once it is established. Allocations must be invariant; the same allocation must be made each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.8.7.

The image itself (pointed to by data) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming a single two-dimensional image slice; and *depth* slices are stacked from back to front. When the final R, G, B,

Base Internal Format	RGBA Values	Internal Components
ALPHA	А	A
LUMINANCE	R	L
LUMINANCE_ALPHA	R,A	L,A
INTENSITY	R	Ι
RGB	R,G,B	R,G,B
RGBA	R,G,B,A	R,G,B,A

Table 3.15: Conversion from RGBA pixel components to internal texture, table, or filter components. See section 3.8.9 for a description of the texture components R, G, B, A, L, and I.

and A components have been computed for a group, they are assigned to components of a *texel* as described by table 3.15. Counting from zero, each resulting Nth texel is assigned internal integer coordinates (i, j, k), where

$$egin{aligned} &i = (N egin{aligned} &\mathrm{mod} \ width) - b_s \ &j = (\lfloor rac{N}{width}
floor egin{aligned} &\mathrm{mod} \ height) - b_s \ &k = (\lfloor rac{N}{width imes height}
floor egin{aligned} &\mathrm{mod} \ depth
floor - b_s \ &k \end{aligned}$$

and b_s is the specified *border* width. Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of k.

Each color component is converted (by rounding to nearest) to a fixedpoint value with n bits, where n is the number of bits of storage allocated to that component in the image array. We assume that the fixed-point representation used represents each value $k/(2^n - 1)$, where $k \in \{0, 1, \ldots, 2^n - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

The *level* argument to **TexImage3D** is an integer *level-of-detail* number. Levels of detail are discussed below, under **Mipmapping**. The main texture image has a level of detail number of 0. If a level-of-detail less than zero is specified, the error INVALID_VALUE is generated.

The *border* argument to **TexImage3D** is a border width. The significance of borders is described below. The border width affects the required dimensions of the texture image: it must be the case that

$$w_s = 2^n + 2b_s \tag{3.11}$$

3.8. TEXTURING

Sized	Base	R	G	В	A	L	Ι
Internal Format	Internal Format	bits	bits	bits	bits	\mathbf{bits}	\mathbf{bits}
ALPHA4	ALPHA				4		
ALPHA8	ALPHA				8		
ALPHA12	ALPHA				12		
ALPHA16	ALPHA				16		
LUMINANCE4	LUMINANCE					4	
LUMINANCE8	LUMINANCE					8	
LUMINANCE12	LUMINANCE					12	
LUMINANCE16	LUMINANCE					16	
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA				4	4	
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA				2	6	
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA				8	8	
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA				4	12	
LUMINANCE12_ALPHA12	LUMINANCE_ALPHA				12	12	
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA				16	16	
INTENSITY4	INTENSITY						4
INTENSITY8	INTENSITY						8
INTENSITY12	INTENSITY						12
INTENSITY16	INTENSITY						16
R3_G3_B2	RGB	3	3	2			
RGB4	RGB	4	4	4			
RGB5	RGB	5	5	5			
RGB8	RGB	8	8	8			
RGB10	RGB	10	10	10			
RGB12	RGB	12	12	12			
RGB16	RGB	16	16	16			
RGBA2	RGBA	2	2	2	2		
RGBA4	RGBA	4	4	4	4		
RGB5_A1	RGBA	5	5	5	1		
RGBA8	RGBA	8	8	8	8		
RGB10_A2	RGBA	10	10	10	2		
RGBA12	RGBA	12	12	12	12		
RGBA16	RGBA	16	16	16	16		

Table 3.16: Correspondence of sized internal formats to base internal formats, and *desired* component resolutions for each sized internal format.

$$h_s = 2^m + 2b_s (3.12)$$

$$d_s = 2^l + 2b_s \tag{3.13}$$

for some integers n, m, and l, where w_s , h_s , and d_s are the specified image width, height, and depth. If any one of these relationships cannot be satisfied, then the error INVALID_VALUE is generated.

Currently, the maximum border width b_t is 1. If b_s is less than zero, or greater than b_t , then the error INVALID_VALUE is generated.

The maximum allowable width, height, or depth of a three-dimensional texture image is an implementation dependent function of the level-of-detail and internal format of the resulting image array. It must be at least $2^{k-lod} + 2b_t$ for image arrays of level-of-detail 0 through k, where k is the log base 2 of MAX_3D_TEXTURE_SIZE, lod is the level-of-detail of the image array, and b_t is the maximum border width. It may be zero for image arrays of any level-of-detail greater than k. The error INVALID_VALUE is generated if the specified image is too large to be stored under any conditions.

In a similar fashion, the maximum allowable width of a one- or twodimensional texture image, and the maximum allowable height of a twodimensional texture image, must be at least $2^{k-lod} + 2b_t$ for image arrays of level 0 through k, where k is the log base 2 of MAX_TEXTURE_SIZE.

Furthermore, an implementation may allow a one-, two-, or threedimensional image array of level 1 or greater to be created only if a complete¹ set of image arrays consistent with the requested array can be supported. Likewise, an implementation may allow an image array of level 0 to be created only if that single image array can be supported.

The command

void TexImage2D(enum target, int level, int internalformat, sizei width, sizei height, int border, enum format, enum type, void *data);

is used to specify a two-dimensional texture image. *target* must be either TEXTURE_2D, or PROXY_TEXTURE_2D in the special case discussed in section 3.8.7. The other parameters match the corresponding parameters of TexImage3D.

¹For this purpose the definition of "complete", as provided under **Mipmapping**, is augmented as follows: 1) it is as though **TEXTURE_BASE_LEVEL** is 0 and **TEXTURE_MAX_LEVEL** is 1000. 2) Excluding borders, the dimensions of the next lower numbered array are all understood to be twice the corresponding dimensions of the specified array.

For the purposes of decoding the texture image, **TexImage2D** is equivalent to calling **TexImage3D** with corresponding arguments and *depth* of 1, except that

- The *depth* of the image is always 1 regardless of the value of *border*.
- Convolution will be performed on the image (possibly changing its *width* and *height*) if SEPARABLE_2D or CONVOLUTION_2D is enabled.
- UNPACK_SKIP_IMAGES is ignored.

Finally, the command

```
void TexImage1D( enum target, int level,
    int internalformat, sizei width, int border,
    enum format, enum type, void *data );
```

is used to specify a one-dimensional texture image. *target* must be either TEXTURE_1D, or PROXY_TEXTURE_1D in the special case discussed in section 3.8.7.)

For the purposes of decoding the texture image, **TexImage1D** is equivalent to calling **TexImage2D** with corresponding arguments and *height* of 1, except that

- The *height* of the image is always 1 regardless of the value of *border*.
- Convolution will be performed on the image (possibly changing its *width*) only if CONVOLUTION_1D is enabled.

An image with zero width, height (**TexImage2D** and **TexImage3D** only), or depth (**TexImage3D** only) indicates the null texture. If the null texture is specified for the level-of-detail specified by **TEXTURE_BASE_LEVEL**, it is as if texturing were disabled.

The image indicated to the GL by the image pointer is decoded and copied into the GL's internal memory. This copying effectively places the decoded image inside a border of the maximum allowable width b_t whether or not a border has been specified (see figure 3.10)². If no border or a border smaller than the maximum allowable width has been specified, then the image is still stored as if it were surrounded by a border of the maximum possible width. Any excess border (which surrounds the specified image,

²Figure 3.10 needs to show a three-dimensional texture image.

including any border) is assigned unspecified values. A two-dimensional texture has a border only at its left, right, top, and bottom ends, and a one-dimensional texture has a border only at its left and right ends.

We shall refer to the (possibly border augmented) decoded image as the *texture array*. A three-dimensional texture array has width, height, and depth

$$w_t = 2^n + 2b_t$$
$$h_t = 2^m + 2b_t$$
$$d_t = 2^l + 2b_t$$

where b_t is the maximum allowable border width and n, m, and l are defined in equations 3.11, 3.12, and 3.13. A two-dimensional texture array has depth $d_t = 1$, with height h_t and width w_t as above, and a one-dimensional texture array has depth $d_t = 1$, height $h_t = 1$, and width w_t as above.

An element (i, j, k) of the texture array is called a *texel* (for a twodimensional texture, k is irrelevant; for a one-dimensional texture, j and k are both irrelevant). The *texture value* used in texturing a fragment is determined by that fragment's associated (s, t, r) coordinates, but may not correspond to any actual texel. See figure 3.10.

If the *data* argument of **TexImage1D**, **TexImage2D**, or **TexImage3D** is a null pointer (a zero-valued pointer in the C implementation), a one-, two-, or three-dimensional texture array is created with the specified *target*, *level*, *internalformat*, *width*, *height*, and *depth*, but with unspecified image contents. In this case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the *data* pointer were valid.

3.8.2 Alternate Texture Image Specification Commands

Two-dimensional and one-dimensional texture images may also be specified using image data taken directly from the framebuffer, and rectangular subregions of existing texture images may be respecified.

The command

```
void CopyTexImage2D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    sizei height, int border );
```

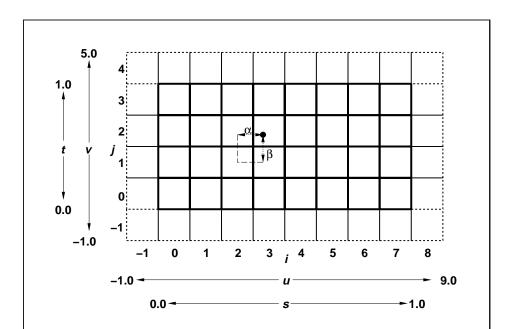


Figure 3.10. A texture image and the coordinates used to access it. This is a two-dimensional texture with n = 3 and m = 2. A one-dimensional texture would consist of a single horizontal strip. α and β , values used in blending adjacent texels to obtain a texture value, are also shown.

defines a two-dimensional texture array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. Currently, target must be TEXTURE_2D. x, y, width, and *height* correspond precisely to the corresponding arguments to **Copy**-**Pixels** (refer to section 4.3.3); they specify the image's width and height, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels**, with argument *type* set to COLOR, stopping after pixel transfer processing is complete. Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, and A values from the resulting pixel groups. Parameters level, internal format, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage2D**, except that *internal format* may not be specified as 1, 2, 3, or 4. An invalid value specified for internal format generates the error INVALID_ENUM. The constraints on width, height, and border are exactly those for the equivalent arguments of **TexImage2D**.

The command

```
void CopyTexImage1D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    int border );
```

defines a one-dimensional texture array in exactly the manner of **TexIm-age1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently, *target* must be **TEXTURE_1D**. For the purposes of decoding the texture image, **CopyTexImage1D** is equivalent to calling **CopyTexImage2D** with corresponding arguments and *height* of 1, except that the *height* of the image is always 1, regardless of the value of *border*. *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage1D**, except that *internalformat* may not be specified as 1, 2, 3, or 4. The constraints on width and *border* are exactly those of the equivalent arguments of **TexImage1D**.

Six additional commands,

```
void TexSubImage3D( enum target, int level, int xoffset,
    int yoffset, int zoffset, sizei width, sizei height,
    sizei depth, enum format, enum type, void *data );
void TexSubImage2D( enum target, int level, int xoffset,
    int yoffset, sizei width, sizei height, enum format,
    enum type, void *data );
```

- void TexSubImage1D(enum target, int level, int xoffset, sizei width, enum format, enum type, void *data);
- void CopyTexSubImage3D(enum target, int level, int xoffset, int yoffset, int zoffset, int x, int y, sizei width, sizei height);
- void CopyTexSubImage2D(enum target, int level, int xoffset, int yoffset, int x, int y, sizei width, sizei height);

respecify only a rectangular subregion of an existing texture array. No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texture array, nor is any change made to texel values outside the specified subregion. Currently the *target* arguments of **TexSubImage1D** and **CopyTexSubImage1D** must be **TEXTURE_1D**, the *target* arguments of **TexSubImage2D** and **CopyTexSubImage2D** must be **TEXTURE_2D**, and the *target* arguments of **TexSubImage3D** and **Copy-TexSubImage3D** must be **TEXTURE_3D**. The *level* parameter of each command specifies the level of the texture array that is modified. If *level* is less than zero or greater than the base 2 logarithm of the maximum texture width or height, the error **INVALID_VALUE** is generated.

TexSubImage3D arguments width, height, depth, format, type, and data match the corresponding arguments to **TexImage3D**, meaning that they are specified using the same values, and have the same meanings. Likewise, **TexSubImage2D** arguments width, height, format, type, and data match the corresponding arguments to **TexImage2D**, and **TexSubImage1D** arguments width, format, type, and data match the corresponding arguments to **TexImage1D**.

CopyTexSubImage3D and **CopyTexSubImage2D** arguments x, y, width, and *height* match the corresponding arguments to **CopyTexImage2D**³. **CopyTexSubImage1D** arguments x, y, and width match the corresponding arguments to **CopyTexImage1D**. Each of the **TexSubImage** commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, and A pixel group values to the texture components is controlled by the *internalformat* of the texture array, not by an argument to the command.

 $^{{}^{3}}$ Because the frame buffer is inherently two-dimensional, there is no **CopyTexIm-age3D** command.

Arguments *xoffset*, *yoffset*, and *zoffset* of **TexSubImage3D** and **Copy-TexSubImage3D** specify the lower left texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texture array. The *depth* argument associated with **CopyTexSubImage3D** is always 1, because framebuffer memory is two-dimensional - only a portion of a single s, t slice of a three-dimensional texture is replaced by **CopyTexSubImage3D**.

Negative values of *xoffset*, *yoffset*, and *zoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s , h_s , d_s , and b_s to be the specified width, height, depth, and border width of the texture array, (not the actual array dimensions w_t , h_t , d_t , and b_t), and taking x, y, z, w, h, and d to be the *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* argument values, any of the following relationships generates the error **INVALID_VALUE**:

$$egin{aligned} x < -b_s \ x+w > w_s-b_s \ y < -b_s \ y+h > h_s-b_s \ z < -b_s \ z+d > d_s-b_s \end{aligned}$$

(Recall that d_s , w_s , and h_s include twice the specified border width b_s .) Counting from zero, the *n*th pixel group is assigned to the texel with internal integer coordinates [i, j, k], where

$$i = x + (n \mod w)$$
$$j = y + (\lfloor \frac{n}{w} \rfloor \mod h)$$
$$k = z + (\lfloor \frac{n}{width * height} \rfloor \mod d$$

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTex-SubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texture array. Negative values of *xoffset* and *yoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s , h_s , and b_s to be the specified width, height, and border width of the texture array, (not the actual array dimensions w_t , h_t , and b_t), and taking x, y, w, and h to be the *xoffset*, *yoffset*, *width*, and

height argument values, any of the following relationships generates the error INVALID_VALUE:

$$egin{aligned} &x < -b_s \ &x + w > w_s - b_s \ &y < -b_s \ &y + h > h_s - b_s \end{aligned}$$

(Recall that w_s and h_s include twice the specified border width b_s .) Counting from zero, the *n*th pixel group is assigned to the texel with internal integer coordinates [i, j], where

$$i = x + (n \mod w)$$
$$j = y + (\lfloor \frac{n}{w} \rfloor \mod h)$$

The *xoffset* argument of **TexSubImage1D** and **CopyTexSubImage1D** specifies the left texel coordinate of a *width*-wide subregion of the texture array. Negative values of *xoffset* correspond to the coordinates of border texels. Taking w_s and b_s to be the specified width and border width of the texture array, and x and w to be the *xoffset* and *width* argument values, either of the following relationships generates the error INVALID_VALUE:

$$x < -b_s$$

 $x + w > w_s - b_s$

Counting from zero, the *n*th pixel group is assigned to the texel with internal integer coordinates [i], where

$$i = x + (n \bmod w)$$

3.8.3 Texture Parameters

Various parameters control how the texture array is treated when applied to a fragment. Each parameter is set by calling

```
void TexParameter{if}( enum target, enum pname,
   T param );
void TexParameter{if}v( enum target, enum pname,
   T params );
```

Name	Type	Legal Values	
TEXTURE_WRAP_S	integer	CLAMP, CLAMP_TO_EDGE, REPEAT	
TEXTURE_WRAP_T	integer	CLAMP, CLAMP_TO_EDGE, REPEAT	
TEXTURE_WRAP_R	integer	CLAMP, CLAMP_TO_EDGE, REPEAT	
TEXTURE_MIN_FILTER	integer	NEAREST,	
		LINEAR,	
		NEAREST_MIPMAP_NEAREST,	
		NEAREST_MIPMAP_LINEAR,	
		LINEAR_MIPMAP_NEAREST,	
		LINEAR_MIPMAP_LINEAR,	
TEXTURE_MAG_FILTER	integer	NEAREST,	
		LINEAR	
TEXTURE_BORDER_COLOR	4 floats	any 4 values in $[0, 1]$	
TEXTURE_PRIORITY	float	any value in $[0, 1]$	
TEXTURE_MIN_LOD	float	any value	
TEXTURE_MAX_LOD	float	any value	
TEXTURE_BASE_LEVEL	integer	any non-negative integer	
TEXTURE_MAX_LEVEL	integer	any non-negative integer	

Table 3.17: Texture parameters and their values.

target is the target, either TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D. pname is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.17. In the first form of the command, param is a value to which to set a single-valued parameter; in the second form of the command, params is an array of parameters whose type depends on the parameter being set. If the values for TEXTURE_BORDER_COLOR are specified as integers, the conversion for signed integers from table 2.6 is applied to convert the values to floating-point. Each of the four values set by TEXTURE_BORDER_COLOR is clamped to lie in [0, 1].

3.8.4 Texture Wrap Modes

If TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R is set to REPEAT, then the GL ignores the integer part of s, t, or r coordinates, respectively, using only the fractional part. (For a number f, the fractional part is $f - \lfloor f \rfloor$, regardless of the sign of f; recall that the *floor* function truncates towards $-\infty$.) CLAMP causes s, t, or r coordinates to be clamped to the range [0, 1]. The initial state is for all of s, t, and r behavior to be that given by REPEAT.

CLAMP_TO_EDGE clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. The color returned when clamping is derived only from texels at the edge of the texture image.

Texture coordinates are clamped to the range [min, max]. The minimum value is defined as

$$min = \frac{1}{2N}$$

where N is the size of the one-, two-, or three-dimensional texture image in the direction of clamping. The maximum value is defined as

$$max = 1 - min$$

so that clamping is always symmetric about the [0, 1] mapped range of a texture coordinate.

3.8.5 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image.

Scale Factor and Level of Detail

The choice is governed by a scale factor $\rho(x, y)$ and the *level of detail* parameter $\lambda(x, y)$, defined as

$$\lambda'(x, y) = \log_2[\rho(x, y)]$$

$$\lambda = \begin{cases} \text{TEXTURE_MAX_LOD}, & \lambda' > \text{TEXTURE_MAX_LOD} \\ \lambda', & \text{TEXTURE_MIN_LOD} \le \lambda' \le \text{TEXTURE_MAX_LOD} \\ \text{TEXTURE_MIN_LOD}, & \lambda' < \text{TEXTURE_MIN_LOD} \\ undefined, & \text{TEXTURE_MIN_LOD} > \text{TEXTURE_MAX_LOD} \end{cases}$$
(3.14)

If $\lambda(x, y)$ is less than or equal to the constant c (described below in section 3.8.6) the texture is said to be magnified; if it is greater, the texture is minified.

The initial values of TEXTURE_MIN_LOD and TEXTURE_MAX_LOD are chosen so as to never clamp the normal range of λ . They may be respecified for a specific texture by calling **TexParameter**[if].

Let s(x, y) be the function that associates an s texture coordinate with each set of window coordinates (x, y) that lie within a primitive; define t(x, y) and r(x, y) analogously. Let $u(x, y) = 2^n s(x, y)$, $v(x, y) = 2^m t(x, y)$, and $w(x, y) = 2^l r(x, y)$, where n, m, and l are as defined by equations 3.11, 3.12, and 3.13 with w_s , h_s , and d_s equal to the width, height, and depth of the image array whose level is **TEXTURE_BASE_LEVEL**. For a one-dimensional texture, define $v(x, y) \equiv 0$ and $w(x, y) \equiv 0$; for a two-dimensional texture, define $w(x, y) \equiv 0$. For a polygon, ρ is given at a fragment with window coordinates (x, y) by

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2}\right\}$$
(3.15)

where $\partial u/\partial x$ indicates the derivative of u with respect to window x, and similarly for the other derivatives.

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2}/l,$$
(3.16)

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$ with (x_1, y_1) and (x_2, y_2) being the segment's window coordinate endpoints and $l = \sqrt{\Delta x^2 + \Delta y^2}$. For a point, pixel rectangle, or bitmap, $\rho \equiv 1$.

While it is generally agreed that equations 3.15 and 3.16 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal ρ with a function f(x, y) subject to these conditions:

- 1. f(x,y) is continuous and monotonically increasing in each of $|\partial u/\partial x|$, $|\partial u/\partial y|$, $|\partial v/\partial x|$, $|\partial v/\partial y|$, $|\partial w/\partial x|$, and $|\partial w/\partial y|$
- $2. \ Let$

$$m_u = \max\left\{ \left| \frac{\partial u}{\partial x} \right|, \left| \frac{\partial u}{\partial y} \right| \right\}$$

$$m_v = \max\left\{ \left| \frac{\partial v}{\partial x} \right|, \left| \frac{\partial v}{\partial y} \right|
ight\}$$

 $m_w = \max\left\{ \left| \frac{\partial w}{\partial x} \right|, \left| \frac{\partial w}{\partial y} \right|
ight\}.$

Then $\max\{m_u, m_v, m_w\} \le f(x, y) \le m_u + m_v + m_w$.

When λ indicates minification, the value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected. When TEXTURE_MIN_FILTER is NEAREST, the texel in the image array of level TEXTURE_BASE_LEVEL that is nearest (in Manhattan distance) to that specified by (s, t, r) is obtained. This means the texel at location (i, j, k) becomes the texture value, with *i* given by

$$i = \begin{cases} \lfloor u \rfloor, & s < 1\\ 2^n - 1, & s = 1 \end{cases}$$
(3.17)

(Recall that if TEXTURE_WRAP_S is REPEAT, then $0 \le s < 1$.) Similarly, j is found as

$$j = \begin{cases} \lfloor v \rfloor, & t < 1\\ 2^m - 1, & t = 1 \end{cases}$$
(3.18)

and k is found as

$$k = \begin{cases} \lfloor w \rfloor, & r < 1\\ 2^{l} - 1, & r = 1 \end{cases}$$
(3.19)

For a one-dimensional texture, j and k are irrelevant; the texel at location i becomes the texture value. For a two-dimensional texture, k is irrelevant; the texel at location (i, j) becomes the texture value.

When TEXTURE_MIN_FILTER is LINEAR, a $2 \times 2 \times 2$ cube of texels in the image array of level TEXTURE_BASE_LEVEL is selected. This cube is obtained by first clamping texture coordinates as described above under Texture Wrap Modes (if the wrap mode for a coordinate is CLAMP or CLAMP_TO_EDGE) and computing

$$i_0 = \begin{cases} \lfloor u - 1/2 \rfloor \mod 2^n, & \texttt{TEXTURE_WRAP_S is REPEAT} \\ \lfloor u - 1/2 \rfloor, & otherwise \end{cases}$$

$$j_0 = \left\{ egin{array}{cccc} \lfloor v - 1/2
floor \mod 2^m, & { t texture_wrap_t is repeat} \ \lfloor v - 1/2
floor, & otherwise \end{array}
ight.$$

 and

$$k_0 = \left\{ egin{array}{ccc} \lfloor w - 1/2
floor \mod 2^l, & { t texture_wrap_r is repeat} \ \lfloor w - 1/2
floor, & otherwise \end{array}
ight.$$

Then

$$i_1 = \begin{cases} (i_0 + 1) \mod 2^n, & \texttt{TEXTURE_WRAP_S} \text{ is REPEAT} \\ i_0 + 1, & otherwise \end{cases}$$

$$j_1 = \begin{cases} (j_0 + 1) \mod 2^m, & \texttt{TEXTURE_WRAP_T is REPEAT} \\ j_0 + 1, & otherwise \end{cases}$$

 and

$$k_1 = \begin{cases} (k_0 + 1) \mod 2^l, & \texttt{TEXTURE_WRAP_R is REPEAT} \\ k_0 + 1, & otherwise \end{cases}$$

Let

$$lpha = \operatorname{frac}(u - 1/2)$$

 $eta = \operatorname{frac}(v - 1/2)$
 $\gamma = \operatorname{frac}(w - 1/2)$

where $\operatorname{frac}(x)$ denotes the fractional part of x.

For a three-dimensional texture, the texture value τ is found as

$$\begin{aligned} \tau &= (1-\alpha)(1-\beta)(1-\gamma)\tau_{i_0j_0k_0} + \alpha(1-\beta)(1-\gamma)\tau_{i_1j_0k_0} \\ &+ (1-\alpha)\beta(1-\gamma)\tau_{i_0j_1k_0} + \alpha\beta(1-\gamma)\tau_{i_1j_1k_0} \\ &+ (1-\alpha)(1-\beta)\gamma\tau_{i_0j_0k_1} + \alpha(1-\beta)\gamma\tau_{i_1j_0k_1} \\ &+ (1-\alpha)\beta\gamma\tau_{i_0j_1k_1} + \alpha\beta\gamma\tau_{i_1j_1k_1} \end{aligned}$$

where τ_{ijk} is the texel at location (i,j,k) in the three-dimensional texture image.

For a two-dimensional texture,

$$\tau = (1 - \alpha)(1 - \beta)\tau_{i_0j_0} + \alpha(1 - \beta)\tau_{i_1j_0} + (1 - \alpha)\beta\tau_{i_0j_1} + \alpha\beta\tau_{i_1j_1} \quad (3.20)$$

where τ_{ij} is the texel at location (i, j) in the two-dimensional texture image.

And for a one-dimensional texture,

$$\tau = (1 - \alpha)\tau_{i_0} + \alpha\tau_{i_1}$$

where τ_i is the texel at location *i* in the one-dimensional texture.

If any of the selected τ_{ijk} , τ_{ij} , or τ_i in the above equations refer to a border texel with $i < -b_s$, $j < -b_s$, $k < -b_s$, $i \ge w_s - b_s$, $j \ge h_s - b_s$, or $j \ge d_s - b_s$, then the border color given by the current setting of TEXTURE_BORDER_COLOR is used instead of the unspecified value or values. The RGBA values of the TEXTURE_BORDER_COLOR are interpreted to match the texture's internal format in a manner consistent with table 3.15.

Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a mipmap. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the image array of level TEXTURE_BASE_LEVEL, excluding its border, has dimensions $2^n \times 2^m \times 2^l$, then there are max $\{n, m, l\} + 1$ image arrays in the mipmap. Each array subsequent to the array of level TEXTURE_BASE_LEVEL has dimensions

$$\sigma(i-1) imes \sigma(j-1) imes \sigma(k-1)$$

where the dimensions of the previous array are

$$\sigma(i)\times\sigma(j)\times\sigma(k)$$

 and

$$\sigma(x) = \left\{egin{array}{cc} 2^x & x > 0 \ 1 & x \leq 0 \end{array}
ight.$$

until the last array is reached with dimension $1 \times 1 \times 1$.

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from **TEXTURE_BASE_LEVEL** for the original texture array through $p = \max\{n, m, l\}$ + TEXTURE_BASE_LEVEL with each unit increase indicating an array of half the dimensions of the previous one as already described. If texturing is enabled (and TEXTURE_MIN_FILTER is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays TEXTURE_BASE_LEVEL through $q = \min\{p, \text{TEXTURE_MAX_LEVEL}\}$ is incomplete, then it is as if texture mapping were disabled. The set of arrays TEXTURE_BASE_LEVEL through q is incomplete if the internal formats of all the mipmap arrays were not specified with the same symbolic constant, if the border widths of the mipmap arrays are not the same, if the dimensions of the mipmap arrays do not follow the sequence described above, if TEXTURE_MAX_LEVEL < TEXTURE_BASE_LEVEL, or if TEXTURE_BASE_LEVEL > p. Array levels k where k < TEXTURE_BASE_LEVEL or k > q are insignificant.

The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL may be respecified for a specific texture by calling **TexParameter**[if]. The error INVALID_VALUE is generated if either value is negative.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let c be the value of λ at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of λ where $\lambda > c$). In the following equations, let

 $b = \texttt{TEXTURE_BASE_LEVEL}$

For mipmap filters NEAREST_MIPMAP_NEAREST and LINEAR_MIPMAP_NEAREST, the dth mipmap array is selected, where

$$d = \begin{cases} b, & \lambda \leq \frac{1}{2} \\ \lceil b + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, b + \lambda \leq q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, b + \lambda > q + \frac{1}{2} \end{cases}$$
(3.21)

The rules for NEAREST or LINEAR filtering are then applied to the selected array.

For mipmap filters NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR, the level d_1 and d_2 mipmap arrays are selected, where

$$d_1 = \begin{cases} q, & b+\lambda \ge q\\ \lfloor b+\lambda \rfloor, & otherwise \end{cases}$$
(3.22)

$$d_2 = \begin{cases} q, & b+\lambda \ge q\\ d_1+1, & otherwise \end{cases}$$
(3.23)

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values τ_1 and τ_2 . The final texture value is then found as

$$\tau = [1 - \operatorname{frac}(\lambda)]\tau_1 + \operatorname{frac}(\lambda)\tau_2.$$

3.8.6 Texture Magnification

When λ indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER (equations 3.17, 3.18, and 3.19 are used); LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER (equation 3.20 is used). The level-of-detail TEXTURE_BASE_LEVEL texture array is always used for magnification.

Finally, there is the choice of c, the minification vs. magnification switchover point. If the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then c = 0.5. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise c = 0.

3.8.7 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the three sets of mipmap arrays (one-, two-, and three-dimensional) and their number. Each array has associated with it a width, height (twoor three-dimensional only), and depth (three-dimensional only), a border width, an integer describing the internal format of the image, and six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the image. Each initial texture array is null (zero width, height, and depth, zero border width, internal format 1, with zero-sized components). Next, there are the two sets of texture properties; each consists of the selected minification and magnification filters, the wrap modes for s, t (two- and three-dimensional only), and r (three-dimensional only), the TEXTURE_BORDER_COLOR, two integers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the texture is resident and the priority associated with each set of properties. The value of the resident flag is determined by the GL and may change as a result of other GL operations. The flag may only be queried, not set, by applications. See section 3.8.8). In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR, and the value for TEXTURE_MAG_FILTER is LINEAR. s, t, and r wrap modes are all set to REPEAT.

The values of TEXTURE_MIN_LOD and TEXTURE_MAX_LOD are -1000 and 1000 respectively. The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. TEXTURE_PRIORITY is 1.0, and TEXTURE_BORDER_COLOR is (0,0,0,0). The initial value of TEXTURE_RESIDENT is determined by the GL.

In addition to the one-, two-, and three-dimensional sets of image arrays, partially instantiated one-, two-, and three-dimensional sets of proxy image arrays are maintained. Each proxy array includes width, height (twoand three-dimensional arrays only), depth (three-dimensional arrays only), border width, and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy arrays do not include image data, nor do they include texture properties. When **TexImage3D** is executed with *target* specified as **PROXY_TEXTURE_3D**, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by **TexImage3D** called with *target* set to **TEXTURE_3D**, no error is generated, but the proxy width, height, depth, border width, and component resolutions are set to zero. If the image array would be supported by such a call to **TexImage3D**, the proxy state values are set exactly as though the actual image array were being specified. No pixel data are transferred or processed in either case.

One- and two-dimensional proxy arrays are operated on in the same way when **TexImage1D** is executed with *target* specified as **PROXY_TEXTURE_1D**, or **TexImage2D** is executed with *target* specified as **PROXY_TEXTURE_2D**.

There is no image associated with any of the proxy textures. Therefore **PROXY_TEXTURE_1D**, **PROXY_TEXTURE_2D**, and **PROXY_TEXTURE_3D** cannot be used as textures, and their images must never be queried using **GetTexImage**. The error **INVALID_ENUM** is generated if this is attempted. Likewise, there is no nonlevel-related state associated with a proxy texture, and **GetTex-Parameteriv** or **GetTexParameterfv** may not be called with a proxy texture *target*. The error **INVALID_ENUM** is generated if this is attempted.

3.8.8 Texture Objects

In addition to the default textures TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D named one-, two-, and three-dimensional texture objects can be created and operated upon. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by *binding* an unused name to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D. The binding is effected by calling

3.8. TEXTURING

void BindTexture(enum target, uint texture);

with *target* set to the desired texture target and *texture* set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.8.7, set to the same initial values. If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D respectively, it is and remains a one-, two-, or three-dimensional texture until it is deleted.

BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality than the specified *target*. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to *target* is broken.

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target to which a texture object is bound is enabled, the state of the bound texture object directs the texturing operation.

In the initial state, TEXTURE_1D, TEXTURE_2D, and TEXTURE_3D have one-, two-, and three-dimensional texture state vectors associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-, or threedimensional texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D respectively while 0 is bound to the corresponding targets.

Texture objects are deleted by calling

void DeleteTextures(sizei n, uint *textures);

textures contains n names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to one of the targets TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D is deleted, it is as though **BindTexture** had been executed with the same target and texture zero. Unused names in textures are silently ignored, as is the value zero.

The command

```
void GenTextures( sizei n, uint *textures );
```

returns n previously unused texture object names in *textures*. These names are marked as used, for the purposes of **GenTextures** only, but they acquire texture state and a dimensionality only when they are first bound, just as if they were unused.

An implementation may choose to establish a working set of texture objects on which binding operations are performed with higher performance. A texture object that is currently part of the working set is said to be *resident*. The command

boolean AreTexturesResident(sizei n, uint *textures, boolean *residences);

returns TRUE if all of the *n* texture objects named in *textures* are resident, or if the implementation does not distinguish a working set. If at least one of the texture objects named in *textures* is not resident, then FALSE is returned, and the residence of each texture object is returned in *residences*. Otherwise the contents of *residences* are not changed. If any of the names in *textures* are unused or are zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of *residences* are indeterminate. The residence status of a single bound texture object can also be queried by calling **Get-TexParameteriv** or **GetTexParameterfv** with *target* set to the target to which the texture object is bound, and *pname* set to TEXTURE_RESIDENT.

AreTexturesResident indicates only whether a texture object is currently resident, not whether it could not be made resident. An implementation may choose to make a texture object resident only on first use, for example. The client may guide the GL implementation in determining which texture objects should be resident by specifying a priority for each texture object. The command

sets the priorities of the *n* texture objects named in *textures* to the values in *priorities*. Each priority value is clamped to the range [0,1] before it is assigned. Zero indicates the lowest priority, with the least likelihood of being resident. One indicates the highest priority, with the greatest likelihood of being resident. The priority of a single bound texture object may also be changed by calling **TexParameteri**, **TexParameterf**, **TexParameteriv**, or **TexParameterfv** with *target* set to the target to which the texture object is bound, *pname* set to **TEXTURE_PRIORITY**, and *param* or *params* specifying the new priority value (which is clamped to the range [0,1] before being assigned). **PrioritizeTextures** silently ignores attempts to prioritize unused texture object names or zero (default textures).

3.8.9 Texture Environments and Texture Functions

The command

void TexEnv{if}(enum target, enum pname, T param); void TexEnv{if}v(enum target, enum pname, T params);

sets parameters of the *texture environment* that specifies how texture values are interpreted when texturing a fragment. *target* must currently be the symbolic constant TEXTURE_ENV. *pname* is a symbolic constant indicating the parameter to be set. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is a pointer to an array of parameters: either a single symbolic constant or a value or group of values to which the parameter should be set. The possible environment parameters are TEXTURE_ENV_MODE and TEXTURE_ENV_COLOR. TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, or BLEND; TEXTURE_ENV_COLOR is set to an RGBA color by providing four single-precision floating-point values in the range [0, 1] (values outside this range are clamped to it). If integers are provided for TEXTURE_ENV_COLOR, then they are converted to floating-point as specified in table 2.6 for signed integers.

The value of TEXTURE_ENV_MODE specifies a *texture function*. The result of this function depends on the fragment and the texture array value. The precise form of the function depends on the base internal formats of the texture arrays that were last specified. In the following two tables, R_f , G_f , B_f , and A_f are the primary color components of the incoming fragment; R_t , G_t , B_t , A_t , L_t , and I_t are the filtered texture values; R_c , G_c , B_c , and A_c are the texture environment color values; and R_v , G_v , B_v , and A_v are the primary color components computed by the texture function. All of these color values are in the range [0, 1]. The REPLACE and MODULATE texture functions are specified in table 3.18, and the DECAL and BLEND texture functions are specified in table 3.19.

The state required for the current texture environment consists of the four-valued integer indicating the texture function and four floating-point **TEXTURE_ENV_COLOR** values. In the initial state, the texture function is given by MODULATE and TEXTURE_ENV_COLOR is (0, 0, 0, 0).

Base	REPLACE	MODULATE
Internal Format	Texture Function	Texture Function
ALPHA	$R_v = R_f$	$R_v = R_f$
	$G_v = G_f$	$G_v = G_f$
	$B_v = B_f$	$B_v = B_f$
	$A_v = A_t$	$A_v = A_f A_t$
LUMINANCE	$R_v = L_t$	$R_v = R_f L_t$
(or 1)	$G_v = L_t$	$G_v = G_f L_t$
	$B_v = L_t$	$B_v = B_f L_t$
	$A_v = A_f$	$A_v = A_f$
LUMINANCE_ALPHA	$R_v = L_t$	$R_v = R_f L_t$
(or 2)	$G_v = L_t$	$G_v = G_f L_t$
	$B_v = L_t$	$B_v = B_f L_t$
	$A_v = A_t$	$A_v = A_f A_t$
INTENSITY	$R_v = I_t$	$R_v = R_f I_t$
	$G_v = I_t$	$G_v = G_f I_t$
	$B_v = I_t$	$B_v = B_f I_t$
	$A_v = I_t$	$A_v = A_f I_t$
RGB	$R_v = R_t$	$R_v = R_f R_t$
(or 3)	$G_v = G_t$	$G_v = G_f G_t$
	$B_v = B_t$	$B_v = B_f B_t$
	$A_v = A_f$	$A_v = A_f$
RGBA	$R_v = R_t$	$R_v = R_f R_t$
(or 4)	$G_v = G_t$	$G_v = G_f G_t$
	$B_v = B_t$	$B_v = B_f B_t$
	$A_v = A_t$	$A_v = A_f A_t$

Table 3.18: Replace and modulate texture functions.

Base	DECAL	BLEND
Internal Format	Texture Function	Texture Function
ALPHA	undefined	$R_v = R_f$
		$G_v = G_f$
		$B_v = B_f$
		$A_v = A_f A_t$
LUMINANCE	undefined	$R_v = R_f (1 - L_t) + R_c L_t$
(or 1)		$G_v = G_f(1 - L_t) + G_c L_t$
		$B_v = B_f (1 - L_t) + B_c L_t$
		$A_v = A_f$
LUMINANCE_ALPHA	undefined	$R_v = R_f (1 - L_t) + R_c L_t$
(or 2)		$G_v = G_f(1 - L_t) + G_c L_t$
		$B_v = B_f (1 - L_t) + B_c L_t$
		$A_v = A_f A_t$
INTENSITY	undefined	$R_v = R_f (1 - I_t) + R_c I_t$
		$G_v = G_f(1 - I_t) + G_c I_t$
		$B_v = B_f (1 - I_t) + B_c I_t$
		$A_v = A_f (1 - I_t) + A_c I_t$
RGB	$R_v = R_t$	$R_v = R_f (1 - R_t) + R_c R_t$
(or 3)	$G_v = G_t$	$G_v = G_f(1 - G_t) + G_c G_t$
	$B_v = B_t$	$B_v = B_f (1 - B_t) + B_c B_t$
	$A_v = A_f$	$A_v = A_f$
RGB A	$R_v = R_f(1 - A_t) + R_t A_t$	$R_v = R_f (1 - R_t) + R_c R_t$
(or 4)	$G_v = G_f(1 - A_t) + G_t A_t$	$G_v = G_f(1 - G_t) + G_c G_t$
	$B_v = B_f(1 - A_t) + B_t A_t$	$B_v = B_f (1 - B_t) + B_c B_t$
	$A_v = A_f$	$A_v = A_f A_t$

Table 3.19: Decal and blend texture functions.

3.8.10 Texture Application

Texturing is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D to enable the one-, two-, or three-dimensional texture, respectively. If both two- and one-dimensional textures are enabled, the twodimensional texture is used. If the three-dimensional and either of the two- or one-dimensional textures is enabled, the three-dimensional texture is used. If all texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality using the rules given in sections 3.8.5 and 3.8.6. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment's primary R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

The required state is three bits indicating whether each of one-, two-, or three-dimensional texturing is enabled or disabled. In the initial state, all texturing is disabled.

3.9 Color Sum

At the beginning of color sum, a fragment has two RGBA colors: a primary color \mathbf{c}_{pri} (which texturing, if enabled, may have modified) and a secondary color \mathbf{c}_{sec} . The components of these two colors are summed to produce a single post-texturing RGBA color \mathbf{c} . The components of \mathbf{c} are then clamped to the range [0, 1].

Color sum has no effect in color index mode.

3.10 Fog

If enabled, fog blends a fog color with a rasterized fragment's post-texturing color using a blending factor f. Fog is enabled and disabled with the **Enable** and **Disable** commands using the symbolic constant FOG.

This factor f is computed according to one of three equations:

$$f = \exp(-d \cdot z), \tag{3.24}$$

3.10. FOG

$$f = \exp(-(d \cdot z)^2), \text{ or }$$
(3.25)

$$f = \frac{e-z}{e-s} \tag{3.26}$$

(z is the eye-coordinate distance from the eye, (0, 0, 0, 1) in eye coordinates, to the fragment center). The equation, along with either d or e and s, is specified with

If pname is FOG_MODE, then param must be, or params must point to an integer that is one of the symbolic constants EXP, EXP2, or LINEAR, in which case equation 3.24, 3.25, or 3.26, respectively, is selected for the fog calculation (if, when 3.26 is selected, e = s, results are undefined). If pname is FOG_DENSITY, FOG_START, or FOG_END, then param is or params points to a value that is d, s, or e, respectively. If d is specified less than zero, the error INVALID_VALUE results.

An implementation may choose to approximate the eye-coordinate distance from the eye to each fragment center by $|z_e|$. Further, f need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are.

No matter which equation and approximation is used to compute f, the result is clamped to [0, 1] to obtain the final f.

f is used differently depending on whether the GL is in RGBA or color index mode. In RGBA mode, if C_r represents a rasterized fragment's R, G, or B value, then the corresponding value produced by fog is

$$C = fC_r + (1 - f)C_f.$$

(The rasterized fragment's A value is not changed by fog blending.) The R, G, B, and A values of C_f are specified by calling **Fog** with *pname* equal to **FOG_COLOR**; in this case *params* points to four values comprising C_f . If these are not floating-point values, then they are converted to floating-point using the conversion given in table 2.6 for signed integers. Each component of C_f is clamped to [0, 1] when specified.

In color index mode, the formula for fog blending is

$$I = i_r + (1 - f)i_f$$

where i_r is the rasterized fragment's color index and i_f is a single-precision floating-point value. $(1 - f)i_f$ is rounded to the nearest fixed-point value

139

with the same number of bits to the right of the binary point as i_r , and the integer portion of I is masked (bitwise ANDed) with $2^n - 1$, where n is the number of bits in a color in the color index buffer (buffers are discussed in chapter 4). The value of i_f is set by calling **Fog** with *pname* set to **FOG_INDEX** and *param* being or *params* pointing to a single value for the fog index. The integer part of i_f is masked with $2^n - 1$.

The state required for fog consists of a three valued integer to select the fog equation, three floating-point values d, e, and s, an RGBA fog color and a fog color index, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, FOG_MODE is EXP, d = 1.0, e = 1.0, and s = 0.0; $C_f = (0, 0, 0, 0)$ and $i_f = 0$.

3.11 Antialiasing Application

Finally, if antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. In RGBA mode, the value is multiplied by the fragment's alpha (A) value to yield a final alpha value. In color index mode, the value is used to set the low order bits of the color index value as described in section 3.2.

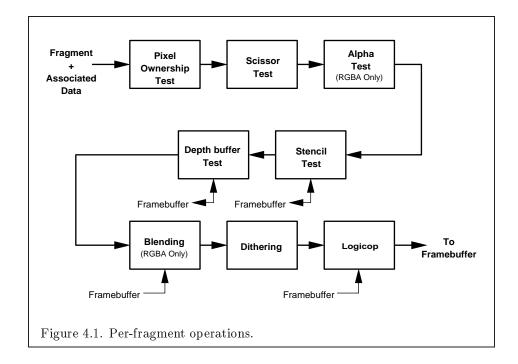
Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer consists of a set of pixels arranged as a two-dimensional array. The height and width of this array may vary from one GL implementation to another. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may also vary depending on the particular GL implementation or context.

Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several *logical buffers*. These are the *color*, *depth*, *stencil*, and *accumulation* buffers. The color buffer actually consists of a number of buffers: the *front left* buffer, the *front right* buffer, the *back left* buffer, the *back right* buffer, and some number of *auxiliary* buffers. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, back buffers, or auxiliary buffers at all. Further, an implementation or context may not provide depth, stencil, or accumulation buffers.

Color buffers consist of either unsigned integer color indices or R, G, B, and, optionally, A unsigned integer values. The number of bitplanes in each of the color buffers, the depth buffer, the stencil buffer, and the accumulation buffer is fixed and window dependent. If an accumulation buffer is provided,



it must have at least as many bitplanes per R, G, and B color component as do the color buffers.

The initial state of all provided bitplanes is undefined.

4.1 Per-Fragment Operations

A fragment produced by rasterization with window coordinates of (x_w, y_w) modifies the pixel in the framebuffer at that location based on a number of parameters and conditions. We describe these modifications and tests, diagrammed in Figure 4.1, in the order in which they are performed. Figure 4.1 diagrams these modifications and tests.

4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location (x_w, y_w) in the framebuffer is currently owned by the GL (more precisely, by this GL context). If it is not, the window system decides the fate the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

4.1.2 Scissor test

The scissor test determines if (x_w, y_w) lies within the scissor rectangle defined by four values. These values are set with

If $left \leq x_w < left + width$ and $bottom \leq y_w < bottom + height$, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using **Enable** or **Disable** using the constant SCISSOR_TEST. When disabled, it is as if the scissor test always passes. If either width or height is less than zero, then the error INVALID_VALUE is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state left = bottom = 0; width and height are determined by the size of the GL window. Initially, the scissor test is disabled.

4.1.3 Alpha test

This step applies only in RGBA mode. In color index mode, proceed to the next step. The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant **ALPHA_TEST**. When disabled, it is as if the comparison always passes. The test is controlled with

void AlphaFunc(enum func, clampf ref);

func is a symbolic constant indicating the alpha test function; ref is a reference value. ref is clamped to lie in [0, 1], and then converted to a fixed-point value according to the rules given for an A component in section 2.13.9. For purposes of the alpha test, the fragment's alpha value is also rounded to the nearest integer. The possible constants specifying the test function are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL, meaning pass the fragment never, always, if the fragment's alpha value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the reference value, respectively. The required state consists of the floating-point reference value, an eightvalued integer indicating the comparison function, and a bit indicating if the comparison is enabled or disabled. The initial state is for the reference value to be 0 and the function to be ALWAYS. Initially, the alpha test is disabled.

4.1.4 Stencil test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (x_w, y_w) and a reference value. The test is controlled with

void StencilFunc(enum func, int ref, uint mask); void StencilOp(enum sfail, enum dpfail, enum dppass);

The test is enabled or disabled with the **Enable** and **Disable** commands, using the symbolic constant **STENCIL_TEST**. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

ref is an integer reference value that is used in the unsigned stencil comparison. It is clamped to the range $[0, 2^s - 1]$, where s is the number of bits in the stencil buffer. func is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL. Accordingly, the stencil test passes never, always, if the reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer. The s least significant bits of mask are bitwise ANDed with both the reference and the stored stencil value. The ANDed values are those that participate in the comparison.

StencilOp takes three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. *sfail* indicates what action is taken if the stencil test fails. The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, and INVERT. These correspond to keeping the current value, setting it to zero, replacing it with the reference value, incrementing it, decrementing it, or bitwise inverting it. For purposes of increment and decrement, the stencil bits are considered as an unsigned integer; values clamp at 0 and the maximum representable value. The same symbolic values are given to indicate the stencil action if the depth buffer test (below) fails (*dpfail*), or if it passes (*dppass*).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** and **StencilOp**, and a bit indicating whether stencil testing is enabled or disabled.

In the initial state, stenciling is disabled, the stencil reference value is zero, the stencil comparison function is ALWAYS, and the stencil *mask* is all ones. Initially, all three stencil operations are KEEP. If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to **StencilOp**.

4.1.5 Depth buffer test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant **DEPTH_TEST**. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and the fragment is passed to the next operation. The stencil value, however, is modified as indicated below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

The comparison is specified with

void DepthFunc(enum func);

This command takes a single symbolic constant: one of NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GREATER, GEQUAL, NOTEQUAL. Accordingly, the depth buffer test passes never, always, if the incoming fragment's z_w value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment's (x_w, y_w) coordinates.

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment's (x_w, y_w) coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment's (x_w, y_w) location is set to the fragment's z_w value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In the initial state the function is LESS and the test is disabled.

If there is no depth buffer, it is as if the depth buffer test always passes.

4.1.6 Blending

Blending combines the incoming fragment's R, G, B, and A values with the R, G, B, and A values stored in the framebuffer at the incoming fragment's (x_w, y_w) location.

This blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode; in color index mode it is bypassed. Blending is enabled or disabled using **Enable** or **Disable** with the symbolic constant **BLEND**. If it is disabled, or if logical operation on color values is enabled (section 4.1.8), proceed to the next stage.

In the following discussion, C_s refers to the source color for an incoming fragment, C_d refers to the destination color at the corresponding framebuffer location, and C_c refers to a constant color in the GL state. Individual RGBA components of these colors are denoted by subscripts of s, d, and crespectively.

Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme given in section 2.13.9 (Final Color Processing), as are source (fragment) components. Constant color components are taken to be floating point values.

Prior to blending, each fixed-point color component undergoes an implied conversion to floating point. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating point.

The commands that control blending are

void BlendFunc(enum src, enum dst);

Using BlendColor

The constant color C_c to be used in blending is specified with **BlendColor**. The four parameters are clamped to the range [0, 1] before being stored. The constant color can be used in both the source and destination blending factors.

BlendColor is an imaging subset feature (see section 3.6.2), and is only allowed when the imaging subset is supported.

Using BlendEquation

Blending capability is defined by the *blend equation*. BlendEquation mode FUNC_ADD defines the blending equation as

$$C = C_s S + C_d D$$

where C_s and C_d are the source and destination colors, and S and D are quadruplets of weighting factors as specified by **BlendFunc**.

If mode is FUNC_SUBTRACT, the blending equation is defined as

$$C = C_s S - C_d D$$

If mode is FUNC_REVERSE_SUBTRACT, the blending equation is defined as

$$C = C_d D - C_s S$$

If *mode* is MIN, the blending equation is defined as

$$C = min(C_s, C_d)$$

Finally, if *mode* is MAX, the blending equation is defined as

$$C = max(C_s, C_d)$$

The blending equation is evaluated separately for each color component and the corresponding weighting factors.

BlendEquation is an imaging subset feature (see section 3.6.2). If the imaging subset is not available, then blending always uses the blending equation FUNC_ADD.

Using BlendFunc

BlendFunc *src* indicates how to compute a source blending factor, while *dst* indicates how to compute a destination factor. The possible arguments and their corresponding computed source and destination factors are summarized in Tables 4.1 and 4.2. Addition or subtraction of quadruplets means adding or subtracting them component-wise.

The computed source and destination blending quadruplets are applied to the source and destination R, G, B, and A values to obtain a new set of values that are sent to the next operation. Let the source and destination blending quadruplets be S and D, respectively. Then a quadruplet of values is computed using the blend equation specified by **BlendEquation**. Each

Value	Blend Factors
ZERO	(0, 0, 0, 0)
ONE	(1, 1, 1, 1)
DST_COLOR	(R_d, G_d, B_d, A_d)
ONE_MINUS_DST_COLOR	$(1,1,1,1) - (R_d,G_d,B_d,A_d)$
SRC_ALPHA	(A_s, A_s, A_s, A_s)
ONE_MINUS_SRC_ALPHA	$(1,1,1,1)-(A_s,A_s,A_s,A_s)$
DST_ALPHA	(A_d, A_d, A_d, A_d)
ONE_MINUS_DST_ALPHA	$(1,1,1,1) - (A_d,A_d,A_d,A_d)$
CONSTANT_COLOR	(R_c, G_c, B_c, A_c)
ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1, 1) - (R_c, G_c, B_c, A_c)$
CONSTANT_ALPHA	(A_c, A_c, A_c, A_c)
ONE_MINUS_CONSTANT_ALPHA	$(1,1,1,1)-(A_c,A_c,A_c,A_c)$
SRC_ALPHA_SATURATE	(f,f,f,1)

Table 4.1: Values controlling the source blending function and the source blending values they compute. $f = \min(A_s, 1 - A_d)$.

Value	Blend factors
ZERO	(0, 0, 0, 0)
ONE	(1, 1, 1, 1)
SRC_COLOR	(R_s, G_s, B_s, A_s)
ONE_MINUS_SRC_COLOR	$(1,1,1,1)-(R_s,G_s,B_s,A_s)$
SRC_ALPHA	(A_s, A_s, A_s, A_s)
ONE_MINUS_SRC_ALPHA	$(1,1,1,1)-(A_s,A_s,A_s,A_s)$
DST_ALPHA	(A_d, A_d, A_d, A_d)
ONE_MINUS_DST_ALPHA	$(1,1,1,1)-(A_d,A_d,A_d,A_d)$
CONSTANT_COLOR	(R_c, G_c, B_c, A_c)
ONE_MINUS_CONSTANT_COLOR	$(1, 1, 1, 1) - (R_c, G_c, B_c, A_c)$
CONSTANT_ALPHA	(A_c, A_c, A_c, A_c)
ONE_MINUS_CONSTANT_ALPHA	$(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)$

Table 4.2: Values controlling the destination blending function and the destination blending values they compute.

floating-point value in this quadruplet is clamped to [0, 1] and converted back to a fixed-point value in the manner described in section 2.13.9. The resulting four values are sent to the next operation.

BlendFunc arguments CONSTANT_COLOR, ONE_MINUS_CONSTANT_COLOR, CONSTANT_ALPHA, and ONE_MINUS_CONSTANT_ALPHA are imaging subset features (see section 3.6.2), and are only allowed when the imaging subset is provided.

Blending State

The state required for blending is an integer indicating the blending equation, two integers indicating the source and destination blending functions, four floating-point values to store the RGBA constant blend color, and a bit indicating whether blending is enabled or disabled. The initial blending equation is FUNC_ADD. The initial blending functions are ONE for the source function and ZERO for the destination function. The initial constant blend color is (R, G, B, A) = (0, 0, 0, 0). Initially, blending is disabled.

Blending occurs once for each color buffer currently enabled for writing (section 4.2.1) using each buffer's color for C_d . If a color buffer has no A value, then A_d is taken to be 1.

4.1.7 Dithering

Dithering selects between two color values or indices. In RGBA mode, consider the value of any of the color components as a fixed-point value with m bits to the left of the binary point, where m is the number of bits allocated to that component in the framebuffer; call each such value c. For each c, dithering selects a value c_1 such that $c_1 \in \{\max\{0, \lceil c \rceil - 1\}, \lceil c \rceil\}$ (after this selection, treat c_1 as a fixed point value in [0,1] with m bits). This selection may depend on the x_w and y_w coordinates of the pixel. In color index mode, the same rule applies with c being a single color index. c must not be larger than the maximum value representable in the framebuffer for either the component or the index, as appropriate.

Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend only the incoming value and the fragment's xand y window coordinates. If dithering is disabled, then each color component is truncated to a fixed-point value with as many bits as there are in the corresponding component in the framebuffer; a color index is rounded to the nearest integer representable in the color index portion of the framebuffer.

Dithering is enabled with Enable and disabled with Disable using the

symbolic constant DITHER. The state required is thus a single bit. Initially, dithering is enabled.

4.1.8 Logical Operation

Finally, a logical operation is applied between the incoming fragment's color or index values and the color or index values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's (x, y) coordinates. The logical operation on color indices is enabled or disabled with **Enable** or **Disable** using the symbolic constant **INDEX_LOGIC_OP**. (For compatibility with GL version 1.0, the symbolic constant LOGIC_OP may also be used.) The logical operation on color values is enabled or disabled with **Enable** or **Disable** using the symbolic constant **COLOR_LOGIC_OP**. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of **BLEND**.

The logical operation is selected by

```
void LogicOp( enum op );
```

op is a symbolic constant; the possible constants and corresponding operations are enumerated in Table 4.3. In this table, s is the value of the incoming fragment and d is the value stored in the framebuffer. The numeric values assigned to the symbolic constants are the same as those assigned to the corresponding symbolic values in the X window system.

Logical operations are performed independently for each color index buffer that is selected for writing, or for each red, green, blue, and alpha value of each color buffer that is selected for writing. The required state is an integer indicating the logical operation, and two bits indicating whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by COPY, and to be disabled.

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting a Buffer for Writing

The first such operation is controlling the buffer into which color values are written. This is accomplished with

Argument value	Operation
CLEAR	0
AND	$s \wedge d$
AND_REVERSE	$s \wedge \neg d$
СОРҮ	s
AND_INVERTED	$ eg s \wedge d$
NOOP	d
XOR	$s \operatorname{xor} d$
OR	$s \lor d$
NOR	$ eg(s \lor d)$
EQUIV	$\neg(s \text{ xor } d)$
INVERT	$\neg d$
OR_REVERSE	$s \vee \neg d$
COPY_INVERTED	$\neg s$
OR_INVERTED	$\neg s \lor d$
NAND	$ eg(s \wedge d)$
SET	all 1's

Table 4.3: Arguments to LogicOp and their corresponding operations.

void DrawBuffer(enum buf);

buf is a symbolic constant specifying zero, one, two, or four buffers for writing. The constants are NONE, FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, FRONT_BACK, LEFT, RIGHT, FRONT_AND_BACK, and AUXO through AUXn, where n+1 is the number of available auxiliary buffers.

The constants refer to the four potentially visible buffers $front_left$, $front_right$, $back_left$, and $back_right$, and to the *auxiliary* buffers. Arguments other than AUXi that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments other than AUXi that omit reference to FRONT or BACK refer to both front and back buffers. AUXi enables drawing only to *auxiliary* buffer *i*. Each AUXi adheres to AUXi = AUXO + *i*. The constants and the buffers they indicate are summarized in Table 4.4. If **DrawBuffer** is is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context, the error INVALID_OPERATION results.

Indicating a buffer or buffers using **DrawBuffer** causes subsequent pixel color value writes to affect the indicated buffers. If more than one color buffer is selected for drawing, blending and logical operations are computed

symbolic	front	front	back	back	aux
constant	left	right	left	right	i
NONE					
FRONT_LEFT	•				
FRONT_RIGHT		•			
BACK_LEFT			•		
BACK_RIGHT				•	
FRONT	٠	٠			
BACK			•	•	
LEFT	٠		•		
RIGHT		٠		•	
FRONT_AND_BACK	•	•	•	•	
AUXi					•

Table 4.4: Arguments to **DrawBuffer** and the buffers that they indicate.

and applied independently for each buffer. Calling **DrawBuffer** with a value of NONE inhibits the writing of color values to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle buffer selection is a set of up to 4 + n bits. 4 bits indicate if the front left buffer, the front right buffer, the back left buffer, or the back right buffer, are enabled for color writing. The other nbits indicate which of the auxiliary buffers is enabled for color writing. In the initial state, the front buffer or buffers are enabled if there are no back buffers; otherwise, only the back buffer or buffers are enabled.

4.2.2 Fine Control of Buffer Updates

Four commands are used to mask the writing of bits to each of the logical framebuffers after all per-fragment operations have been performed. The commands

control the color buffer or buffers (depending on which buffers are currently indicated for writing). The least significant n bits of mask, where n is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode. In RGBA mode, **ColorMask** is used to mask the writing of R, G, B and A values to the color buffer or buffers. r, g, b, and a indicate whether R, G, B, or A values, respectively, are written or not (a value of **TRUE** means that the corresponding value is written). In the initial state, all bits (in color index mode) and all color values (in RGBA mode) are enabled for writing.

The depth buffer can be enabled or disabled for writing z_w values using

```
void DepthMask( boolean mask );
```

If mask is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The command

```
void StencilMask( uint mask );
```

controls the writing of particular bits into the stencil planes. The least significant s bits of mask comprise an integer mask (s is the number of bits in the stencil buffer), just as for **IndexMask**. The initial state is for the stencil plane mask to be all ones.

The state required for the various masking operations is two integers and a bit: an integer for color indices, an integer for stencil values, and a bit for depth values. A set of four bits is also required indicating which color components of an RGBA value should be written. In the initial state, the integer masks are all ones as are the bits controlling depth value and RGBA component writing.

4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

```
void Clear( bitfield buf );
```

is the bitwise OR of a number of values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT,

STENCIL_BUFFER_BIT, and ACCUM_BUFFER_BIT, indicating the buffers currently enabled for color writing, the depth buffer, the stencil buffer, and the accumulation buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If the mask is not a bitwise OR of the specified values, then the error INVALID_VALUE is generated.

sets the clear value for the color buffers in RGBA mode. Each of the specified components is clamped to [0, 1] and converted to fixed-point according to the rules of section 2.13.9.

```
void ClearIndex( float index );
```

sets the clear color index. *index* is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^m - 1$, where m is the number of bits in a color index value stored in the framebuffer.

```
void ClearDepth( clampd d );
```

takes a floating-point value that is clamped to the range [0, 1] and converted to fixed-point according to the rules for a window z value given in section 2.10.1. Similarly,

void ClearStencil(int s);

takes a single integer argument that is the value to which to clear the stencil buffer. s is masked to the number of bitplanes in the stencil buffer.

```
void ClearAccum( float r, float g, float b, float a );
```

takes four floating-point arguments that are the values, in order, to which to set the R, G, B, and A values of the accumulation buffer (see the next section). These values are clamped to the range [-1, 1] when they are specified.

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, and dithering. The masking operations described in the last section (4.2.2) are also effective. If a buffer is not present, then a **Clear** directed at that buffer has no effect.

The state required for clearing is a clear value for each of the color buffer, the depth buffer, the stencil buffer, and the accumulation buffer. Initially, the RGBA color clear value is (0,0,0,0), the clear color index is 0, and the stencil buffer and accumulation buffer clear values are all 0. The depth buffer clear value is initially 1.0.

4.2.4 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for each of R, G, B, and A. The accumulation buffer is controlled exclusively through the use of

void Accum(enum op, float value);

(except for clearing it). *op* is a symbolic constant indicating an accumulation buffer operation, and *value* is a floating-point value to be used in that operation. The possible operations are ACCUM, LOAD, RETURN, MULT, and ADD.

When the scissor test is enabled (section 4.1.2), then only those pixels within the current scissor box are updated by any **Accum** operation; otherwise, all pixels in the window are updated. The accumulation buffer operations apply identically to every affected pixel, so we describe the effect of each operation on an individual pixel. Accumulation buffer values are taken to be signed values in the range [-1, 1]. Using **ACCUM** obtains R, G, B, and A components from the buffer currently selected for reading (section 4.3.2). Each component, considered as a fixed-point value in [0, 1]. (see section 2.13.9), is converted to floating-point. Each result is then multiplied by *value*. The results of this multiplication are then added to the corresponding color component currently in the accumulation buffer, and the resulting color value replaces the current accumulation buffer color value.

The LOAD operation has the same effect as ACCUM, but the computed values replace the corresponding accumulation buffer components rather than being added to them.

The RETURN operation takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by *value*, and clamps the results to the range [0, 1] The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization, except that the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test (section 4.1.2), and dithering (section 4.1.7). Color masking (section 4.2.2) is also applied. The MULT operation multiplies each R, G, B, and A in the accumulation buffer by *value* and then returns the scaled color components to their corresponding accumulation buffer locations. ADD is the same as MULT except that *value* is added to each of the color components.

The color components operated on by **Accum** must be clamped only if the operation is **RETURN**. In this case, a value sent to the enabled color buffers is first clamped to [0, 1]. Otherwise, results are undefined if the result of an operation on a color component is out of the range [-1, 1]. If there is no accumulation buffer, or if the GL is in color index mode, **Accum** generates the error **INVALID_OPERATION**.

No state (beyond the accumulation buffer itself) is required for accumulation buffering.

4.3 Drawing, Reading, and Copying Pixels

Pixels may be written to and read from the framebuffer using the **Draw**-**Pixels** and **ReadPixels** commands. **CopyPixels** can be used to copy a block of pixels from one portion of the framebuffer to another.

4.3.1 Writing to the Stencil Buffer

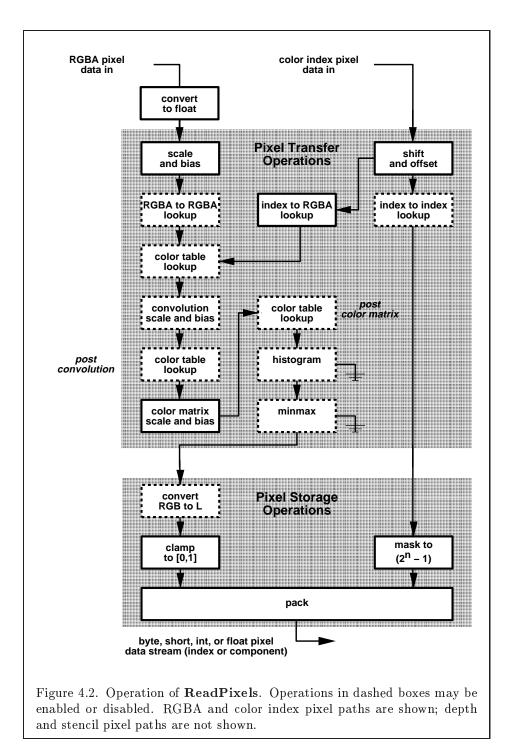
The operation of **DrawPixels** was described in section 3.6.4, except if the format argument was **STENCIL_INDEX**. In this case, all operations described for **DrawPixels** take place, but window (x, y) coordinates, each with the corresponding stencil index, are produced in lieu of fragments. Each coordinatestencil index pair is sent directly to the per-fragment operations, bypassing the texture, fog, and antialiasing application stages of rasterization. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current setting of **StencilMask**.

The error INVALID_OPERATION results if there is no stencil buffer.

4.3.2 Reading Pixels

The method for reading pixels from the framebuffer and placing them in client memory is diagrammed in Figure 4.2. We describe the stages of the pixel reading process in the order in which they occur.

Pixels are read using



157

Parameter Name	Type	Initial Value	Valid Range
PACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
PACK_LSB_FIRST	boolean	FALSE	TRUE / FALSE
PACK_ROW_LENGTH	integer	0	$[0,\infty)$
PACK_SKIP_ROWS	integer	0	$[0,\infty)$
PACK_SKIP_PIXELS	integer	0	$[0,\infty)$
PACK_ALIGNMENT	integer	4	1,2,4,8
PACK_IMAGE_HEIGHT	integer	0	$[0,\infty)$
PACK_SKIP_IMAGES	integer	0	$[0,\infty)$

Table 4.5: **PixelStore** parameters pertaining to **ReadPixels**, **GetTex-Image1D**, **GetTexImage2D**, **GetTexImage3D**, **GetColorTable**, **GetConvolutionFilter**, **GetSeparableFilter**, **GetHistogram**, and **Get-Minmax**.

void ReadPixels(int x, int y, sizei width, sizei height, enum format, enum type, void *data);

The arguments after x and y to **ReadPixels** correspond to those of **Draw-Pixels**. The pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 6.1) are summarized in Table 4.5.

Obtaining Pixels from the Framebuffer

If the *format* is DEPTH_COMPONENT, then values are obtained from the depth buffer. If there is no depth buffer, the error INVALID_OPERATION occurs.

If the *format* is **STENCIL_INDEX**, then values are taken from the stencil buffer; again, if there is no stencil buffer, the error **INVALID_OPERATION** occurs.

For all other formats, the buffer from which values are obtained is one of the color buffers; the selection of color buffer is controlled with **ReadBuffer**. The command

void ReadBuffer(enum src);

takes a symbolic constant as argument. The possible values are FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, FRONT, BACK, LEFT, RIGHT, and AUXO through AUXn. FRONT and LEFT refer to the front left buffer, BACK refers to the back left buffer, and RIGHT refers to the front right buffer. The other constants correspond directly to the buffers that they name. If the requested buffer is missing, then the error INVALID_OPERATION is generated. The initial setting for **ReadBuffer** is FRONT if there is no back buffer and BACK otherwise.

ReadPixels obtains values from the selected buffer from each pixel with lower left hand corner at (x + i, y + j) for $0 \le i < width$ and $0 \le j < height$; this pixel is said to be the *i*th pixel in the *j*th row. If any of these pixels lies outside of the window allocated to the current GL context, the values obtained for those pixels are undefined. Results are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the selected buffer, regardless of how those values were placed there.

If the GL is in RGBA mode, and *format* is one of RED, GREEN, BLUE, ALPHA, RGB, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE_ALPHA, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location. If the framebuffer does not support alpha values then the A that is obtained is 1.0. If *format* is COLOR_INDEX and the GL is in RGBA mode then the error INVALID_OPERATION occurs. If the GL is in color index mode, and *format* is not DEPTH_COMPONENT or STENCIL_INDEX, then the color index is obtained at each pixel location.

Conversion of RGBA values

This step applies only if the GL is in RGBA mode, and then only if *format* is neither STENCIL_INDEX nor DEPTH_COMPONENT. The R, G, B, and A values form a group of elements. Each element is taken to be a fixed-point value in [0, 1] with m bits, where m is the number of bits in the corresponding color component of the selected buffer (see section 2.13.9).

Conversion of Depth values

This step applies only if *format* is DEPTH_COMPONENT. An element is taken to be a fixed-point value in [0,1] with m bits, where m is the number of bits in the depth buffer (see section 2.10.1).

Pixel Transfer Operations

This step is actually the sequence of steps that was described separately in section 3.6.5. After the processing described in that section is completed, groups are processed as described in the following sections.

type Parameter	Index Mask
UNSIGNED_BYTE	$2^8 - 1$
BITMAP	1
BYTE	$2^7 - 1$
UNSIGNED_SHORT	$2^{16} - 1$
SHORT	$2^{15} - 1$
UNSIGNED_INT	$2^{32} - 1$
INT	$2^{31} - 1$

Table 4.6: Index masks used by **ReadPixels**. Floating point data are not masked.

Conversion to L

This step applies only to RGBA component groups, and only if the *format* is either LUMINANCE or LUMINANCE_ALPHA. A value L is computed as

$$L = R + G + B$$

where R, G, and B are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

Final Conversion

For an index, if the *type* is not FLOAT, final conversion consists of masking the index with the value given in Table 4.6; if the *type* is FLOAT, then the integer index is converted to a GL float data value.

For an RGBA color, each component is first clamped to [0, 1]. Then the appropriate conversion formula from table 4.7 is applied to the component.

Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for **DrawPixels**. That is, the *i*th group of the *j*th row (corresponding to the *i*th pixel in the *j*th row) is placed in memory just where the *i*th group of the *j*th row would be taken from for **DrawPixels**. See **Unpacking** under section 3.6.4. The only difference is that the storage mode parameters whose names begin with **PACK_** are used instead of those whose names begin with UNPACK_. If the *format* is **RED**, **GREEN**, **BLUE**, **ALPHA**, or **LUMINANCE**,

type Parameter	GL Data Type	Component
		Conversion Formula
UNSIGNED_BYTE	ubyte	$c = (2^8 - 1)f$
BYTE	byte	$c = [(2^8 - 1)f - 1]/2$
UNSIGNED_SHORT	ushort	$c = (2^{16} - 1)f$
SHORT	short	$c = [(2^{16} - 1)f - 1]/2$
UNSIGNED_INT	uint	$c = (2^{32} - 1)f$
INT	int	$c = [(2^{32} - 1)f - 1]/2$
FLOAT	float	c = f
UNSIGNED_BYTE_3_3_2	ubyte	$c = (2^N - 1)f$
UNSIGNED_BYTE_2_3_3_REV	ubyte	$c = (2^N - 1)f$
UNSIGNED_SHORT_5_6_5	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_5_6_5_REV	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_4_4_4	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_4_4_4_REV	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_5_5_5_1	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_1_5_5_5_REV	ushort	$c = (2^N - 1)f$
UNSIGNED_INT_8_8_8_8	uint	$c = (2^N - 1)f$
UNSIGNED_INT_8_8_8_8_REV	uint	$c = (2^N - 1)f$
UNSIGNED_INT_10_10_10_2	uint	$c = (2^N - 1)f$
UNSIGNED_INT_2_10_10_10_REV	uint	$c = (2^N - 1)f$

Table 4.7: Reversed component conversions - used when component data are being returned to client memory. Color, normal, and depth components are converted from the internal floating-point representation (f) to a datum of the specified GL data type (c) using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (See Table 2.2.) Equations with N as the exponent are performed for each bitfield of the packed data type, with N set to the number of bits in the bitfield.

only the corresponding single element is written. Likewise if the *format* is LUMINANCE_ALPHA, RGB, or BGR, only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

4.3.3 Copying Pixels

CopyPixels transfers a rectangle of pixel values from one region of the framebuffer to another. Pixel copying is diagrammed in Figure 4.3.

void CopyPixels(int x, int y, sizei width, sizei height, enum type);

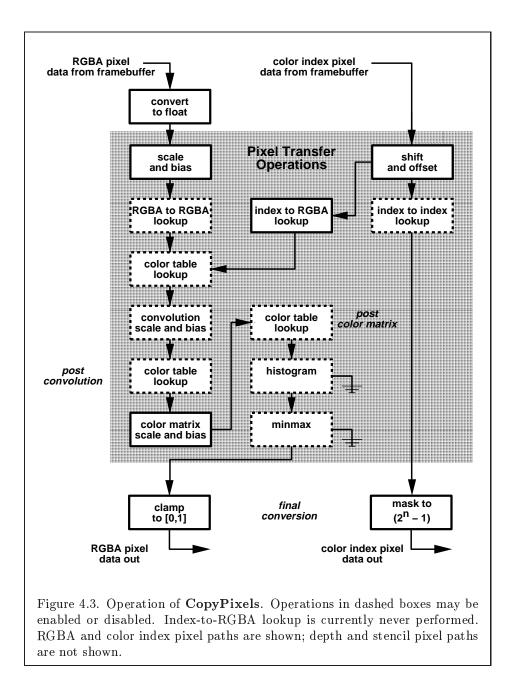
type is a symbolic constant that must be one of COLOR, STENCIL, or DEPTH, indicating that the values to be transferred are colors, stencil values, or depth values, respectively. The first four arguments have the same interpretation as the corresponding arguments to **ReadPixels**.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 3.6.5, just as if **ReadPixels** were called with the corresponding arguments. If the *type* is STENCIL or DEPTH, then it is as if the *format* for **ReadPixels** were **STENCIL_INDEX** or DEPTH_COMPONENT, respectively. If the *type* is COLOR, then if the GL is in RGBA mode, it is as if the *format* were RGBA, while if the GL is in color index mode, it is as if the *format* were COLOR_INDEX.

The groups of elements so obtained are then written to the framebuffer just as if **DrawPixels** had been given *width* and *height*, beginning with final conversion of elements. The effective *format* is the same as that already described.

4.3.4 Pixel Draw/Read state

The state required for pixel operations consists of the parameters that are set with **PixelStore**, **PixelTransfer**, and **PixelMap**. This state has been summarized in Tables 3.1, 3.2, and 3.3. The current setting of **ReadBuffer**, an integer, is also required, along with the current raster position (section 2.12). State set with **PixelStore** is GL client state.



Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of evaluators (used to model curves and surfaces), selection (used to locate rendered primitives on the screen), feedback (which returns GL results before rasterization), display lists (used to designate a group of GL commands for later execution by the GL), flushing and finishing (used to synchronize the GL command stream), and hints.

5.1 Evaluators

Evaluators provide a means to use a polynomial or rational polynomial mapping to produce vertex, normal, and texture coordinates, and colors. The values so produced are sent on to further stages of the GL as if they had been provided directly by the client. Transformations, lighting, primitive assembly, rasterization, and per-pixel operations are not affected by the use of evaluators.

Consider the R^k -valued polynomial $\mathbf{p}(u)$ defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} B_i^n(u) \mathbf{R}_i$$
(5.1)

with $\mathbf{R}_i \in \mathbb{R}^k$ and

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

the *i*th Bernstein polynomial of degree n (recall that $0^0 \equiv 1$ and $\binom{n}{0} \equiv 1$). Each \mathbf{R}_i is a *control point*. The relevant command is

target	k	Values
MAP1_VERTEX_3	3	x, y, z vertex coordinates
MAP1_VERTEX_4	4	x, y, z, w vertex coordinates
MAP1_INDEX	1	color index
MAP1_COLOR_4	4	R, G, B, A
MAP1_NORMAL	3	x, y, z normal coordinates
MAP1_TEXTURE_COORD_1	1	s texture coordinate
MAP1_TEXTURE_COORD_2	2	s, t texture coordinates
MAP1_TEXTURE_COORD_3	3	s, t, r texture coordinates
MAP1_TEXTURE_COORD_4	4	s, t, r, q texture coordinates

Table 5.1: Values specified by the *target* to **Map1**. Values are given in the order in which they are taken.

type is a symbolic constant indicating the range of the defined polynomial. Its possible values, along with the evaluations that each indicates, are given in Table 5.1. order is equal to n + 1; The error INVALID_VALUE is generated if order is less than one or greater than MAX_EVAL_ORDER. points is a pointer to a set of n + 1 blocks of storage. Each block begins with k single-precision floating-point or double-precision floating-point values, respectively. The rest of the block may be filled with arbitrary data. Table 5.1 indicates how k depends on type and what the k values represent in each case.

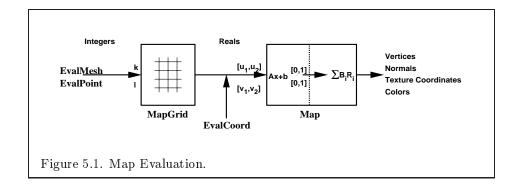
stride is the number of single- or double-precision values (as appropriate) in each block of storage. The error INVALID_VALUE results if stride is less than k. The order of the polynomial, order, is also the number of blocks of storage containing control points.

 u_1 and u_2 give two floating-point values that define the endpoints of the pre-image of the map. When a value u' is presented for evaluation, the formula used is

$$\mathbf{p}'(u') = \mathbf{p}(\frac{u'-u_1}{u_2-u_1}).$$

The error INVALID_VALUE results if $u_1 = u_2$.

Map2 is analogous to Map1, except that it describes bivariate polyno-



mials of the form

$$\mathbf{p}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) \mathbf{R}_{ij}.$$

The form of the **Map2** command is

target is a range type selected from the same group as is used for Map1, except that the string MAP1 is replaced with MAP2. *points* is a pointer to (n+1)(m+1) blocks of storage (*uorder* = n + 1 and *vorder* = m + 1; the error INVALID_VALUE is generated if either *uorder* or *vorder* is less than one or greater than MAX_EVAL_ORDER). The values comprising \mathbf{R}_{ij} are located

$$(ustride)i + (vstride)j$$

values (either single- or double-precision floating-point, as appropriate) past the first value pointed to by *points*. u_1 , u_2 , v_1 , and v_2 define the pre-image rectangle of the map; a domain point (u', v') is evaluated as

$$\mathbf{p}'(u',v') = \mathbf{p}(\frac{u'-u_1}{u_2-u_1},\frac{v'-v_1}{v_2-v_1}).$$

The evaluation of a defined map is enabled or disabled with **Enable** and **Disable** using the constant corresponding to the map as described above. The error INVALID_VALUE results if either *ustride* or *vstride* is less than k, or if u_1 is equal to u_2 , or if v_1 is equal to v_2 .

Figure 5.1 describes map evaluation schematically; an evaluation of enabled maps is effected in one of two ways. The first way is to use

166

void EvalCoord{12}{fd}(T arg); void EvalCoord{12}{fd}v(T arg);

EvalCoord1 causes evaluation of the enabled one-dimensional maps. The argument is the value (or a pointer to the value) that is the domain coordinate, u'. **EvalCoord2** causes evaluation of the enabled two-dimensional maps. The two values specify the two domain coordinates, u' and v', in that order.

When one of the **EvalCoord** commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if a corresponding GL command were issued with the resulting coordinates, with one important difference. The difference is that when an evaluation is performed, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise, the current values are used). The order of the effective commands is immaterial, except that **Vertex** (for vertex coordinate evaluation) must be issued last. Use of evaluators has no effect on the current color, normal, or texture coordinates. If **ColorMaterial** is enabled, evaluated color values affect the result of the lighting equation as if the current color was being modified, but no change is made to the tracking lighting parameters or to the current color.

No command is effectively issued if the corresponding map (of the indicated dimension) is not enabled. If more than one evaluation is enabled for a particular dimension (e.g. MAP1_TEXTURE_COORD_1 and MAP1_TEXTURE_COORD_2), then only the result of the evaluation of the map with the highest number of coordinates is used.

Finally, if either MAP2_VERTEX_3 or MAP2_VERTEX_4 is enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex. Automatic normal generation is controlled with **Enable** and **Disable** with symbolic the constant AUTO_NORMAL. If automatic normal generation is disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used).

For MAP_VERTEX_3, let q = p. For MAP_VERTEX_4, let q = (x/w, y/w, z/w), where (x, y, z, w) = p. Then let

$$\mathbf{m} = rac{\partial \mathbf{q}}{\partial u} imes rac{\partial \mathbf{q}}{\partial v}.$$

Then the generated analytic normal, \mathbf{n} , is given by $\mathbf{n} = \mathbf{m} / \|\mathbf{m}\|$.

The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain. This is done using

```
void \mathbf{MapGrid1}{\mathbf{fd}}( int n, T u_1', T u_2' );
```

for a one-dimensional map or

void
$$MapGrid2{fd}($$
 int n_u , T u'_1 , T u'_2 , int n_v , T v'_1 , T v'_2);

for a two-dimensional map. In the case of **MapGrid1** u'_1 and u'_2 describe an interval, while *n* describes the number of partitions of the interval. The error INVALID_VALUE results if $n \leq 0$. For **MapGrid2**, (u'_1, v'_1) specifies one two-dimensional point and (u'_2, v'_2) specifies another. n_u gives the number of partitions between u'_1 and u'_2 , and n_v gives the number of partitions between v'_1 and v'_2 . If either $n_u \leq 0$ or $n_v \leq 0$, then the error INVALID_VALUE occurs.

Once a grid is defined, an evaluation on a rectangular subset of that grid may be carried out by calling

void **EvalMesh1**(enum *mode*, int p_1 , int p_2);

mode is either POINT or LINE. The effect is the same as performing the following code fragment, with $\Delta u' = (u'_2 - u'_1)/n$:

```
Begin(type);

for i = p_1 to p_2 step 1.0

EvalCoord1(i * \Delta u' + u'_1);

End();
```

where **EvalCoord1f** or **EvalCoord1d** is substituted for **EvalCoord1** as appropriate. If mode is POINT, then type is POINTS; if mode is LINE, then type is LINE_STRIP. The one requirement is that if either i = 0 or i = n, then the value computed from $i * \Delta u' + u'_1$ is precisely u'_1 or u'_2 , respectively.

The corresponding commands for two-dimensional maps are

```
void EvalMesh2( enum mode, int p_1, int p_2, int q_1,
int q_2);
```

168

mode must be FILL, LINE, or POINT. When mode is FILL, then these commands are equivalent to the following, with $\Delta u' = (u'_2 - u'_1)/n$ and $\Delta v' = (v'_2 - v'_1)/m$:

```
for i = q_1 to q_2 - 1 step 1.0

Begin(QUAD_STRIP);

for j = p_1 to p_2 step 1.0

EvalCoord2(j * \Delta u' + u'_1, i * \Delta v' + v'_1);

EvalCoord2(j * \Delta u' + u'_1, (i+1) * \Delta v' + v'_1);

End();
```

If *mode* is LINE, then a call to **EvalMesh2** is equivalent to

```
for i = q_1 to q_2 step 1.0

Begin(LINE_STRIP);

for j = p_1 to p_2 step 1.0

EvalCoord2(j * \Delta u' + u'_1, i * \Delta v' + v'_1);

End();;

for i = p_1 to p_2 step 1.0

Begin(LINE_STRIP);

for j = q_1 to q_2 step 1.0

EvalCoord2(i * \Delta u' + u'_1, j * \Delta v' + v'_1);

End();
```

If mode is POINT, then a call to **EvalMesh2** is equivalent to

```
Begin (POINTS);

for i = q_1 to q_2 step 1.0

for j = p_1 to p_2 step 1.0

EvalCoord2(j * \Delta u' + u'_1, i * \Delta v' + v'_1);

End();
```

Again, in all three cases, there is the requirement that $0 * \Delta u' + u'_1 = u'_1$, $n * \Delta u' + u'_1 = u'_2$, $0 * \Delta v' + v'_1 = v'_1$, and $m * \Delta v' + v'_1 = v'_2$.

An evaluation of a single point on the grid may also be carried out:

void EvalPoint1(int p);

Calling it is equivalent to the command

EvalCoord1 $(p * \Delta u' + u'_1)$;

with $\Delta u'$ and u'_1 defined as above.

void EvalPoint2(int p, int q);

is equivalent to the command

EvalCoord2($p * \Delta u' + u'_1$, $q * \Delta v' + v'_1$);

The state required for evaluators potentially consists of 9 onedimensional map specifications and 9 two-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points, and a set of two values (for a one-dimensional map) or four values (for a two-dimensional map) to describe the domain. The maximum possible order, for either u or v, is implementation dependent (one maximum applies to both u and v), but must be at least 8. Each control point consists of between one and four floating-point values (depending on the type of the map). Initially, all maps have order 1 (making them constant maps). All vertex coordinate maps produce the coordinates (0, 0, 0, 1)(or the appropriate subset); all normal coordinate maps produce (0, 0, 1); RGBA maps produce (1, 1, 1, 1); color index maps produce 1.0; texture coordinate maps produce (0, 0, 0, 1); In the initial state, all maps are disabled. A flag indicates whether or not automatic normal generation is enabled for two-dimensional maps. In the initial state, automatic normal generation is disabled. Also required are two floating-point values and an integer number of grid divisions for the one-dimensional grid specification and four floatingpoint values and two integer grid divisions for the two-dimensional grid specification. In the initial state, the bounds of the domain interval for 1-D is 0 and 1.0, respectively; for 2-D, they are (0,0) and (1.0, 1.0), respectively. The number of grid divisions is 1 for 1-D and 1 in both directions for 2-D. If any evaluation command is issued when no vertex map is enabled, nothing happens.

5.2 Selection

Selection is used by a programmer to determine which primitives are drawn into some region of a window. The region is defined by the current modelview and perspective matrices.

Selection works by returning an array of integer-valued *names*. This array represents the current contents of the *name stack*. This stack is controlled with the commands

void InitNames(void); void PopName(void); void PushName(uint name); void LoadName(uint name);

InitNames empties (clears) the name stack. **PopName** pops one name off the top of the name stack. **PushName** causes *name* to be pushed onto the name stack. LoadName replaces the value on the top of the stack with *name*. Loading a name onto an empty stack generates the error INVALID_OPERATION. Popping a name off of an empty stack generates STACK_UNDERFLOW; pushing a name onto a full stack generates STACK_OVERFLOW. The maximum allowable depth of the name stack is implementation dependent but must be at least 64.

In selection mode, no fragments are rendered into the framebuffer. The GL is placed in selection mode with

int RenderMode(enum mode);

mode is a symbolic constant: one of RENDER, SELECT, or FEEDBACK. RENDER is the default, corresponding to rendering as described until now. SELECT specifies selection mode, and FEEDBACK specifies feedback mode (described below). Use of any of the name stack manipulation commands while the GL is not in selection mode has no effect.

Selection is controlled using

```
void SelectBuffer( sizei n, uint *buffer );
```

buffer is a pointer to an array of unsigned integers (called the selection array) to be potentially filled with names, and n is an integer indicating the maximum number of values that can be stored in that array. Placing the GL in selection mode before **SelectBuffer** has been called results in an error of INVALID_OPERATION as does calling **SelectBuffer** while in selection mode.

In selection mode, if a point, line, polygon, or the valid coordinates produced by a **RasterPos** command intersects the clip volume (section 2.11) then this primitive (or **RasterPos** command) causes a selection *hit*. In the case of polygons, no hit occurs if the polygon would have been culled, but selection is based on the polygon itself, regardless of the setting of **PolygonMode**. When in selection mode, whenever a name stack manipulation command is executed or **RenderMode** is called and there has been a hit since the last time the stack was manipulated or **RenderMode** was called, then a *hit record* is written into the selection array. A hit record consists of the following items in order: a non-negative integer giving the number of elements on the name stack at the time of the hit, a minimum depth value, a maximum depth value, and the name stack with the bottommost element first. The minimum and maximum depth values are the minimum and maximum taken over all the window coordinate z values of each (post-clipping) vertex of each primitive that intersects the clipping volume since the last hit record was written. The minimum and maximum (each of which lies in the range [0, 1]) are each multiplied by $2^{32} - 1$ and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record. No depth offset arithmetic (section 3.5.5) is performed on these values.

Hit records are placed in the selection array by maintaining a pointer into that array. When selection mode is entered, the pointer is initialized to the beginning of the array. Each time a hit record is copied, the pointer is updated to point at the array element after the one into which the topmost element of the name stack was stored. If copying the hit record into the selection array would cause the total number of values to exceed n, then as much of the record as fits in the array is written and an overflow flag is set.

Selection mode is exited by calling **RenderMode** with an argument value other than SELECT. Whenever **RenderMode** is called in selection mode, it returns the number of hit records copied into the selection array and resets the **SelectBuffer** pointer to its last specified value. Values are not guaranteed to be written into the selection array until **RenderMode** is called. If the selection array overflow flag was set, then **RenderMode** returns -1 and clears the overflow flag. The name stack is cleared and the stack pointer reset whenever **RenderMode** is called.

The state required for selection consists of the address of the selection array and its maximum size, the name stack and its associated pointer, a minimum and maximum depth value, and several flags. One flag indicates the current **RenderMode** value. In the initial state, the GL is in the **RENDER** mode. Another flag is used to indicate whether or not a hit has occurred since the last name stack manipulation. This flag is reset upon entering selection mode and whenever a name stack manipulation takes place. One final flag is required to indicate whether the maximum number of copied names would have been exceeded. This flag is reset upon entering selection mode. This flag, the address of the selection array, and its maximum size are GL client state.

5.3 Feedback

Feedback, like selection, is a GL mode. The mode is selected by calling **RenderMode** with FEEDBACK. When the GL is in feedback mode, no fragments are written to the framebuffer. Instead, information about primitives that would have been rasterized is fed back to the application using the GL.

Feedback is controlled using

void FeedbackBuffer(sizei n, enum type, float *buffer);

buffer is a pointer to an array of floating-point values into which feedback information will be placed, and n is a number indicating the maximum number of values that can be written to that array. type is a symbolic constant describing the information to be fed back for each vertex (see Figure 5.2). The error INVALID_OPERATION results if the GL is placed in feedback mode before a call to **FeedbackBuffer** has been made, or if a call to **FeedbackBuffer** is made while in feedback mode.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to **DrawPixels** or **CopyPixels**, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback array, with subsequent blocks following. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling (section 3.5.1) and **PolygonMode** interpretation of polygons (section 3.5.4) has taken place. It may also occur after polygons with more than three edges are broken up into triangles (if the GL implementation renders polygons by performing this decomposition). x, y, and zcoordinates returned by feedback are window coordinates; if w is returned, it is in clip coordinates. No depth offset arithmetic (section 3.5.5) is performed on the z values. In the case of bitmaps and pixel rectangles, the coordinates returned are those of the current raster position.

The texture coordinates and colors returned are these resulting from the clipping operations described in Section 2.13.8. The colors returned are the primary colors.

The ordering rules for GL command interpretation also apply in feedback mode. Each command must be fully interpreted and its effects on both GL

Туре	$\operatorname{coordinates}$	color	texture	total values
2D	x, y	_	_	2
3D	x, y, z	_	-	3
3D_COLOR	x, y, z	k	-	3+k
3D_COLOR_TEXTURE	x, y, z	k	4	7+k
4D_COLOR_TEXTURE	x,y,z,w	k	4	8+k

Table 5.2: Correspondence of feedback type to number of values per vertex. k is 1 in color index mode and 4 in RGBA mode.

state and the values to be written to the feedback buffer completed before a subsequent command may be executed.

The GL is taken out of feedback mode by calling **RenderMode** with an argument value other than **FEEDBACK**. When called while in feedback mode, **RenderMode** returns the number of values placed in the feedback array and resets the feedback array pointer to be *buffer*. The return value never exceeds the maximum number of values passed to **FeedbackBuffer**.

If writing a value to the feedback buffer would cause more values to be written than the specified maximum number of values, then the value is not written and an overflow flag is set. In this case, **RenderMode** returns -1 when it is called, after which the overflow flag is reset. While in feedback mode, values are not guaranteed to be written into the feedback buffer before **RenderMode** is called.

Figure 5.2 gives a grammar for the array produced by feedback. Each primitive is indicated with a unique identifying value followed by some number of vertices. A vertex is fed back as some number of floating-point values determined by the feedback *type*. Table 5.2 gives the correspondence between feedback *buffer* and the number of values returned for each vertex.

The command

```
void PassThrough( float token );
```

may be used as a marker in feedback mode. *token* is returned as if it were a primitive; it is indicated with its own unique identifying value. The ordering of any **PassThrough** commands with respect to primitive specification is maintained by feedback. **PassThrough** may not occur between **Begin** and **End**. It has no effect when the GL is not in feedback mode.

The state required for feedback is the pointer to the feedback array, the maximum number of values that may be placed there, and the feedback *type*.

An overflow flag is required to indicate whether the maximum allowable number of feedback values has been written; initially this flag is cleared. These state variables are GL client state. Feedback also relies on the same mode flag as selection to indicate whether the GL is in feedback, selection, or normal rendering mode.

5.4 Display Lists

A display list is simply a group of GL commands and arguments that has been stored for subsequent execution. The GL may be instructed to process a particular display list (possibly repeatedly) by providing a number that uniquely specifies it. Doing so causes the commands within the list to be executed just as if they were given normally. The only exception pertains to commands that rely upon client state. When such a command is accumulated into the display list (that is, when issued, not when executed), the client state in effect at that time applies to the command. Only server state is affected when the command is executed. As always, pointers which are passed as arguments to commands are dereferenced when the command is issued. (Vertex array pointers are dereferenced when the command s **ArrayElement**, **DrawArrays**, or **DrawElements** are accumulated into a display list.)

A display list is begun by calling

```
void NewList( uint n, enum mode );
```

n is a positive integer to which the display list that follows is assigned, and *mode* is a symbolic constant that controls the behavior of the GL during display list creation. If *mode* is COMPILE, then commands are not executed as they are placed in the display list. If *mode* is COMPILE_AND_EXECUTE then commands are executed as they are encountered, then placed in the display list. If n = 0, then the error INVALID_VALUE is generated.

After calling **NewList** all subsequent GL commands are placed in the display list (in the order the commands are issued) until a call to

```
void EndList( void );
```

occurs, after which the GL returns to its normal command execution state. It is only when **EndList** occurs that the specified display list is actually associated with the index indicated with **NewList**. The error INVALID_OPERATION is generated if **EndList** is called without a previous matching **NewList**,

feedback-list:	
feedback-item feedback-list	pixel-rectangle:
feed back-item	DRAW_PIXEL_TOKEN vertex
	COPY_PIXEL_TOKEN vertex
feedback-item:	passthrough:
point	PASS_THROUGH_TOKEN f
line-segment	
$\operatorname{polygon}$	vertex:
bitmap	2D:
pixel-rectangle	f f
${ m passthrough}$	3D:
	$f \ f \ f$
point:	3D_COLOR:
POINT_TOKEN vertex	f f f color
line-segment:	3D_COLOR_TEXTURE:
LINE_TOKEN vertex vertex	f f f color tex
LINE_RESET_TOKEN vertex vertex	4D_COLOR_TEXTURE:
polygon:	f f f f f color tex
POLYGON_TOKEN $n { m polygon-spec}$	
polygon-spec:	color:
polygon-spec vertex	f f f f f
vertex vertex vertex	f
bitmap:	
BITMAP_TOKEN vertex	tex:
	f f f f f

Figure 5.2: Feedback syntax. f is a floating-point number. n is a floatingpoint integer giving the number of vertices in a polygon. The symbols ending with _TOKEN are symbolic floating-point constants. The labels under the "vertex" rule show the different data returned for vertices depending on the feedback *type*. LINE_TOKEN and LINE_RESET_TOKEN are identical except that the latter is returned only when the line stipple is reset for that line segment. or if **NewList** is called a second time before calling **EndList**. The error **OUT_OF_MEMORY** is generated if **EndList** is called and the specified display list cannot be stored because insufficient memory is available. In this case GL implementations of revision 1.1 or greater insure that no change is made to the previous contents of the display list, if any, and that no other change is made to the GL state, except for the state changed by execution of GL commands when the display list mode is COMPILE_AND_EXECUTE.

Once defined, a display list is executed by calling

```
void CallList( uint n );
```

n gives the index of the display list to be called. This causes the commands saved in the display list to be executed, in order, just as if they were issued without using a display list. If n = 0, then the error INVALID_VALUE is generated.

The command

void CallLists(sizei n, enum type, void *lists);

provides an efficient means for executing a number of display lists. *n* is an integer indicating the number of display lists to be called, and *lists* is a pointer that points to an array of offsets. Each offset is constructed as determined by *lists* as follows. First, *type* may be one of the constants BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, or FLOAT indicating that the array pointed to by *lists* is an array of bytes, unsigned bytes, shorts, unsigned shorts, integers, unsigned integers, or floats, respectively. In this case each offset is found by simply converting each array element to an integer (floating point values are truncated). Further, *type* may be one of 2_BYTES, 3_BYTES, or 4_BYTES, indicating that the array contains sequences of 2, 3, or 4 unsigned bytes, in which case each integer offset is constructed according to the following algorithm:

 $offset \leftarrow 0$ for i = 1 to b $offset \leftarrow offset$ shifted left 8 bits $offset \leftarrow offset + byte$ advance to next byte in the array

b is 2, 3, or 4, as indicated by type. If n = 0, CallLists does nothing.

Each of the n constructed offsets is taken in order and added to a display list base to obtain a display list number. For each number, the indicated display list is executed. The base is set by calling void ListBase(uint base);

to specify the offset.

Indicating a display list index that does not correspond to any display list has no effect. **CallList** or **CallLists** may appear inside a display list. (If the *mode* supplied to **NewList** is **COMPILE_AND_EXECUTE**, then the appropriate lists are executed, but the **CallList** or **CallLists**, rather than those lists' constituent commands, is placed in the list under construction.) To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation dependent limit is placed on the nesting level of display lists during display list execution. This limit must be at least 64.

Two commands are provided to manage display list indices.

```
uint GenLists( sizei s );
```

returns an integer n such that the indices $n, \ldots, n + s - 1$ are previously unused (i.e. there are s previously unused display list indices starting at n). **GenLists** also has the effect of creating an empty display list for each of the indices $n, \ldots, n + s - 1$, so that these indices all become used. **GenLists** returns 0 if there is no group of s contiguous previously unused display list indices, or if s = 0.

boolean IsList(uint list);

returns TRUE if *list* is the index of some display list.

A contiguous group of display lists may be deleted by calling

```
void DeleteLists( uint list, sizei range );
```

where *list* is the index of the first display list to be deleted and *range* is the number of display lists to be deleted. All information about the display lists is lost, and the indices become unused. Indices to which no display list corresponds are ignored. If range = 0, nothing happens.

Certain commands, when called while compiling a display list, are not compiled into the display list but are executed immediately. These are: IsList, GenLists, DeleteLists, FeedbackBuffer, SelectBuffer, RenderMode, VertexPointer, NormalPointer, ColorPointer, Index-Pointer, TexCoordPointer, EdgeFlagPointer, InterleavedArrays, EnableClientState, DisableClientState, PushClientAttrib, Pop-ClientAttrib, ReadPixels, PixelStore, GenTextures, DeleteTextures, AreTexturesResident, IsTexture, Flush, Finish, as well as IsEnabled and all of the Get commands (see Chapter 6).

TexImage3D, TexImage2D, TexImage1D, Histogram, and ColorTable are executed immediately when called with the corresponding proxy arguments PROXY_TEXTURE_3D, PROXY_TEXTURE_2D, PROXY_TEXTURE_1D, PROXY_HISTOGRAM, and PROXY_COLOR_TABLE, PROXY_POST_CONVOLUTION_COLOR_TABLE, or PROXY_POST_COLOR_MATRIX_COLOR_TABLE.

Display lists require one bit of state to indicate whether a GL command should be executed immediately or placed in a display list. In the initial state, commands are executed immediately. If the bit indicates display list creation, an index is required to indicate the current display list being defined. Another bit indicates, during display list creation, whether or not commands should be executed as they are compiled into the display list. One integer is required for the current **ListBase** setting; its initial value is zero. Finally, state must be maintained to indicate which integers are currently in use as display list indices. In the initial state, no indices are in use.

5.5 Flush and Finish

The command

void Flush(void);

indicates that all commands that have previously been sent to the GL must complete in finite time.

The command

void Finish(void);

forces all previous GL commands to complete. **Finish** does not return until all effects from previously issued commands on GL client and server state and the framebuffer are fully realized.

5.6 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

void Hint(enum target, enum hint);

target is a symbolic constant indicating the behavior to be controlled, and hint is a symbolic constant indicating what type of behavior is desired. target may be one of PERSPECTIVE_CORRECTION_HINT, indicating the desired quality of parameter interpolation; POINT_SMOOTH_HINT, indicating the desired sampling quality of points; LINE_SMOOTH_HINT, indicating the desired sampling quality of lines; POLYGON_SMOOTH_HINT, indicating the desired sampling quality of polygons; and FOG_HINT, indicating whether fog calculations are done per pixel or per vertex. hint must be one of FASTEST, indicating that the most efficient option should be chosen; NICEST, indicating that the highest quality option should be chosen; and DONT_CARE, indicating no preference in the matter.

The interpretation of hints is implementation dependent. An implementation may ignore them entirely.

The initial value of all hints is DONT_CARE.

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

6.1 Querying GL State

6.1.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of **Get** commands. There are four commands for obtaining simple state variables:

void GetBooleanv(enum value, boolean *data); void GetIntegerv(enum value, int *data); void GetFloatv(enum value, float *data); void GetDoublev(enum value, double *data);

The commands obtain boolean, integer, floating-point, or double-precision state variables. *value* is a symbolic constant indicating the state variable to return. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data. In addition

boolean IsEnabled(enum value);

can be used to determine if value is currently enabled (as with **Enable**) or disabled.

181

6.1.2 Data Conversions

If a **Get** command is issued that returns value types different from the type of the value being obtained, a type conversion is performed. If **Get-Booleanv** is called, a floating-point or integer value converts to FALSE if and only if it is zero (otherwise it converts to TRUE). If GetIntegerv (or any of the **Get** commands below) is called, a boolean value is interpreted as either 1 or 0, and a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a **DepthRange** value, a depth buffer clear value, or a normal coordinate. In these cases, the **Get** command converts the floating-point value to an integer according the INT entry of Table 4.7; a value not in [-1, 1] converts to an undefined value. If **GetFloatv** is called, a boolean value is interpreted as either 1.0 or 0.0, an integer is coerced to floating-point, and a double-precision floating-point value is converted to single-precision. Analogous conversions are carried out in the case of **GetDoublev**. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order *n* followed by *f*. Similarly, points for evaluator maps are returned in the order that they appeared when passed to **Map1**. **Map2** returns \mathbf{R}_{ij} in the [(uorder)i + j]th block of values (see page 166 for *i*, *j*, *uorder*, and \mathbf{R}_{ij}).

6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category (clip plane, light, material, etc.) as well as a symbolic constant. These are

- void GetClipPlane(enum plane, double eqn[4]);
- void GetLight{if}v(enum light, enum value, T data);
- void GetMaterial{if}v(enum face, enum value, T data);
- void GetTexEnv{if}v(enum env, enum value, T data);
- void GetTexGen{if}v(enum coord, enum value, T data);
- void GetTexParameter{if}v(enum target, enum value, T data);
- void GetTexLevelParameter{if}v(enum target, int lod, enum value, T data);

void GetPixelMap{ui us f}v(enum map, T data); void GetMap{ifd}v(enum map, enum value, T data);

GetClipPlane always returns four double-precision values in *eqn*; these are the coefficients of the plane equation of *plane* in eye coordinates (these coordinates are those that were computed when the plane was specified).

GetLight places information about *value* (a symbolic constant) for *light* (also a symbolic constant) in *data*. POSITION or SPOT_DIRECTION returns values in eye coordinates (again, these are the coordinates that were computed when the position or direction was specified).

GetMaterial, GetTexGen, GetTexEnv, and GetTexParameter are similar to GetLight, placing information about *value* for the target indicated by their first argument into *data*. The *face* argument to GetMaterial must be either FRONT or BACK, indicating the front or back material, respectively. The *env* argument to GetTexEnv must currently be TEXTURE_ENV. The *coord* argument to GetTexGen must be one of S, T, R, or Q. For Get-TexGen, EYE_LINEAR coefficients are returned in the eye coordinates that were computed when the plane was specified; OBJECT_LINEAR coefficients are returned in object coordinates.

GetTexParameter and GetTexLevelParameter parameter target may be one of TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D, indicating the currently bound one-, two-, or three-dimensional texture object. For GetTexLevelParameter, target may also be one of PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, or PROXY_TEXTURE_3D, indicating the one-, two-, or threedimensional proxy state vector. value is a symbolic value indicating which texture parameter is to be obtained. The *lod* argument to GetTexLevelParameter determines which level-of-detail's state is re-If the *lod* argument is less than zero or if it is larger than turned. the maximum allowable level-of-detail then the error INVALID_VALUE occurs. Queries of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, and TEXTURE_INTENSITY_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. Queries of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, and TEXTURE_BORDER return the width, height, depth, and border as specified when the image array was created. The internal format of the image array is queried as TEXTURE_INTERNAL_FORMAT, or as TEXTURE_COMPONENTS for compatibility with GL version 1.0.

For GetPixelMap, the map must be a map name from Table 3.3. For GetMap, map must be one of the map types described in section 5.1, and

value must be one of ORDER, COEFF, or DOMAIN.

6.1.4 Texture Queries

The command

```
void GetTexImage( enum tex, int lod, enum format,
    enum type, void *img );
```

is used to obtain texture images. It is somewhat different from the other get commands; *tex* is a symbolic value indicating which texture is to be obtained. **TEXTURE_1D** indicates a one-dimensional texture, **TEXTURE_2D** indicates a two-dimensional texture, and **TEXTURE_3D** indicates a three-dimensional texture. *lod* is a level-of-detail number, *format* is a pixel format from Table 3.6, *type* is a pixel type from Table 3.5, and *img* is a pointer to a block of memory.

GetTexImage obtains component groups from a texture image with the indicated level-of-detail. The components are assigned among R, G, B, and A according to Table 6.1, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for threedimensional textures. These groups are then packed and placed in client memory. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are applied.

For three-dimensional textures, pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values PACK_IMAGE_HEIGHT and PACK_SKIP_IMAGES are applied. The correspondence of texels to memory locations is as defined for **TexImage3D** in section 3.8.1.

The row length, number of rows, image depth, and number of images are determined by the size of the texture image (including any borders). Calling **GetTexImage** with *lod* less than zero or larger than the maximum allowable causes the error INVALID_VALUE. Calling **GetTexImage** with *format* of COLOR_INDEX, STENCIL_INDEX, or DEPTH_COMPONENT causes the error INVALID_ENUM.

The command

boolean IsTexture(uint texture);

returns TRUE if *texture* is the name of a texture object. If *texture* is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, **IsTexture** returns **FALSE**. A name returned by **GenTextures**, but not yet bound, is not the name of a texture object.

184

Base Internal Format	R	G	В	Α
ALPHA	0	0	0	A_i
LUMINANCE (or 1)	L_i	0	0	1
LUMINANCE_ALPHA (or 2)	L_i	0	0	A_i
INTENSITY	I_i	0	0	1
RGB (or 3)	R_i	G_i	B_i	1
RGBA (or 4)	R_i	G_i	B_i	A_i

Table 6.1: Texture, table, and filter return values. R_i , G_i , B_i , A_i , L_i , and I_i are components of the internal format that are assigned to pixel values R, G, B, and A. If a requested pixel value is not present in the internal format, the specified constant value is used.

6.1.5 Stipple Query

The command

```
void GetPolygonStipple( void *pattern );
```

obtains the polygon stipple. The pattern is packed into memory according to the procedure given in section 4.3.2 for **ReadPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were **BITMAP**, and the *format* were **COLOR_INDEX**.

6.1.6 Color Matrix Query

The scale and bias variables are queried using **GetFloatv** with *pname* set to the appropriate variable name. The top matrix on the color matrix stack is returned by **GetFloatv** called with *pname* set to COLOR_MATRIX. The depth of the color matrix stack, and the maximum depth of the color matrix stack, are queried with **GetIntegerv**, setting *pname* to COLOR_MATRIX_STACK_DEPTH and MAX_COLOR_MATRIX_STACK_DEPTH respectively.

6.1.7 Color Table Query

The current contents of a color table are queried using

```
void GetColorTable( enum target, enum format, enum type,
void *table );
```

target must be one of the regular color table names listed in table 3.4. format and type accept the same values as do the corresponding parameters of **GetTexImage**. The one-dimensional color table image is returned to client memory starting at table. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are performed. Color components that are requested in the specified format, but which are not included in the internal format of the color lookup table, are returned as zero. The assignments of internal color components to the components requested by format are described in Table 6.1.

The functions

void GetColorTableParameter{if}v(enum target, enum pname, T params);

are used for integer and floating point query.

target must be one of the regular or proxy color table names listed in table 3.4. pname is one of COLOR_TABLE_SCALE, COLOR_TABLE_BIAS, COLOR_TABLE_FORMAT, COLOR_TABLE_WIDTH, COLOR_TABLE_RED_SIZE, COLOR_TABLE_GREEN_SIZE, COLOR_TABLE_BLUE_SIZE, COLOR_TABLE_ALPHA_SIZE, COLOR_TABLE_LUMINANCE_SIZE, or COLOR_TABLE_INTENSITY_SIZE. The value of the specified parameter is returned in params.

6.1.8 Convolution Query

The current contents of a convolution filter image are queried with the command

void GetConvolutionFilter(enum target, enum format, enum type, void *image);

target must be CONVOLUTION_1D or CONVOLUTION_2D. format and type accept the same values as do the corresponding parameters of **GetTexImage**. The one-dimensional or two-dimensional images is returned to client memory starting at *image*. Pixel processing and component mapping are identical to those of **GetTexImage**.

The current contents of a separable filter image are queried using

void GetSeparableFilter(enum target, enum format, enum type, void *row, void *column, void *span); target must be SEPARABLE_2D. format and type accept the same values as do the corresponding parameters of **GetTexImage**. The row and column images are returned to client memory starting at row and column respectively. span is currently unused. Pixel processing and component mapping are identical to those of **GetTexImage**.

The functions

```
void GetConvolutionParameter{if}v( enum target,
    enum pname, T params );
```

are used for integer and floating point query. target must he CONVOLUTION_1D, CONVOLUTION_2D, pname or SEPARABLE_2D. of CONVOLUTION_BORDER_COLOR, CONVOLUTION_BORDER_MODE, is one CONVOLUTION_FILTER_SCALE, CONVOLUTION_FILTER_BIAS, CONVOLUTION_FORMAT, CONVOLUTION_HEIGHT, MAX_CONVOLUTION_WIDTH, CONVOLUTION_WIDTH, or MAX_CONVOLUTION_HEIGHT. The value of the specified parameter is returned in params.

6.1.9 Histogram Query

The current contents of the histogram table are queried using

void GetHistogram(enum target, boolean reset, enum format, enum type, void* values);

target must be HISTOGRAM. type and format accept the same values as do the corresponding parameters of **GetTexImage**. The one-dimensional histogram table image is returned to values. Pixel processing and component mapping are identical to those of **GetTexImage**.

If *reset* is **TRUE**, then all counters of all elements of the histogram are reset to zero. Counters are reset whether returned or not.

No counters are modified if *reset* is FALSE.

Calling

void ResetHistogram(enum target);

resets all counters of all elements of the histogram table to zero. *target* must be **HISTOGRAM**.

It is not an error to reset or query the contents of a histogram table with zero entries.

The functions

```
void GetHistogramParameter{if}v( enum target,
    enum pname, T params );
```

are used for integer and floating point query. *target* must be HISTOGRAM or PROXY_HISTOGRAM. *pname* is one of HISTOGRAM_FORMAT, HISTOGRAM_WIDTH, HISTOGRAM_RED_SIZE, HISTOGRAM_GREEN_SIZE, HISTOGRAM_BLUE_SIZE, HISTOGRAM_ALPHA_SIZE, or HISTOGRAM_LUMINANCE_SIZE. *pname* may be HISTOGRAM_SINK only for *target* HISTOGRAM. The value of the specified parameter is returned in *params*.

6.1.10 Minmax Query

The current contents of the minmax table are queried using

```
void GetMinmax( enum target, boolean reset,
    enum format, enum type, void* values );
```

target must be MINMAX. type and format accept the same values as do the corresponding parameters of **GetTexImage**. A one-dimensional image of width 2 is returned to values. Pixel processing and component mapping are identical to those of **GetTexImage**.

If *reset* is **TRUE**, then each minimum value is reset to the maximum representable value, and each maximum value is reset to the minimum representable value. All values are reset, whether returned or not.

No values are modified if reset is FALSE. Calling

void ResetMinmax(enum target);

resets all minimum and maximum values of *target* to to their maximum and minimum representable values, respectively, *target* must be MINMAX.

The functions

```
void GetMinmaxParameter{if}v( enum target,
    enum pname, T params );
```

are used for integer and floating point query. *target* must be MINMAX. *pname* is MINMAX_FORMAT or MINMAX_SINK. The value of the specified parameter is returned in *params*.

6.1.11 Pointer and String Queries

The command

void GetPointerv(enum pname, void **params);

obtains the pointer or pointers named *pname* in the array *params*. The possible values for *pname* are SELECTION_BUFFER_POINTER, FEEDBACK_BUFFER_POINTER, VERTEX_ARRAY_POINTER, NORMAL_ARRAY_POINTER, COLOR_ARRAY_POINTER, INDEX_ARRAY_POINTER, TEXTURE_COORD_ARRAY_POINTER, and EDGE_FLAG_ARRAY_POINTER. Each returns a single pointer value.

Finally,

```
ubyte *GetString( enum name );
```

returns a pointer to a static string describing some aspect of the current GL connection. The possible values for *name* are VENDOR, RENDERER, VERSION, and EXTENSIONS. The format of the RENDERER and VERSION strings is implementation dependent. The EXTENSIONS string contains a space separated list of extension names (The extension names themselves do not contain any spaces); the VERSION string is laid out as follows:

<version number><space><vendor-specific information>

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The vendor specific information is optional. However, if it is present then it pertains to the server and the format and contents are implementation dependent.

GetString returns the version number (returned in the VERSION string) and the extension names (returned in the EXTENSIONS string) that can be supported on the connection. Thus, if the client and server support different versions and/or extensions, a compatible version and list of extensions is returned.

6.1.12 Saving and Restoring State

Besides providing a means to obtain the values of state variables, the GL also provides a means to save and restore groups of state variables. The **PushAttrib**, **PushClientAttrib**, **PopAttrib** and **PopClientAttrib** commands are used for this purpose. The commands

```
void PushAttrib( bitfield mask );
void PushClientAttrib( bitfield mask );
```

take a bitwise OR of symbolic constants indicating which groups of state variables to push onto an attribute stack. **PushAttrib** uses a server attribute stack while **PushClientAttrib** uses a client attribute stack. Each constant refers to a group of state variables. The classification of each variable into a group is indicated in the following tables of state variables. The error STACK_OVERFLOW is generated if **PushAttrib** or **PushClientAttrib** is executed while the corresponding stack depth is MAX_ATTRIB_STACK_DEPTH or MAX_CLIENT_ATTRIB_STACK_DEPTH respectively. The commands

```
void PopAttrib( void );
void PopClientAttrib( void );
```

reset the values of those state variables that were saved with the last corresponding **PushAttrib** or **PopClientAttrib**. Those not saved remain unchanged. The error **STACK_UNDERFLOW** is generated if **PopAttrib** or **Pop-ClientAttrib** is executed while the respective stack is empty.

Table 6.2 shows the attribute groups with their corresponding symbolic constant names and stacks.

When **PushAttrib** is called with **TEXTURE_BIT** set, the priorities, border colors, filter modes, and wrap modes of the currently bound texture objects, as well as the current texture bindings and enables, are pushed onto the attribute stack. (Unbound texture objects are not pushed or restored.) When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture objects' priorities, border colors, filter modes, and wrap modes are restored to their pushed values.

The depth of each attribute stack is implementation dependent but must be at least 16. The state required for each attribute stack is potentially 16 copies of each state variable, 16 masks indicating which groups of variables are stored in each stack entry, and an attribute stack pointer. In the initial state, both attribute stacks are empty.

In the tables that follow, a type is indicated for each variable. Table 6.3 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with all matrices, where only the top entry on the stack is returned; with clip planes, where only the selected clip plane is returned, with parameters describing lights, where only the value pertaining

Stack	Attribute	Constant
server	accum-buffer	ACCUM_BUFFER_BIT
server	color-buffer	COLOR_BUFFER_BIT
server	$\operatorname{current}$	CURRENT_BIT
server	depth-buffer	DEPTH_BUFFER_BIT
server	enable	ENABLE_BIT
server	eval	EVAL_BIT
server	\log	FOG_BIT
server	hint	HINT_BIT
server	lighting	LIGHTING_BIT
server	line	LINE_BIT
server	list	LIST_BIT
server	$_{\rm pixel}$	PIXEL_MODE_BIT
server	point	POINT_BIT
server	$\operatorname{polygon}$	POLYGON_BIT
server	polygon-stipple	POLYGON_STIPPLE_BIT
server	scissor	SCISSOR_BIT
server	$\operatorname{stencil-buffer}$	STENCIL_BUFFER_BIT
server	texture	TEXTURE_BIT
server	$\operatorname{transform}$	TRANSFORM_BIT
server	viewport	VIEWPORT_BIT
server		ALL_ATTRIB_BITS
client	vertex-array	CLIENT_VERTEX_ARRAY_BIT
client	pixel-store	CLIENT_PIXEL_STORE_BIT
client	select	can't be pushed or pop'd
client	feedback	can't be pushed or pop'd
client		ALL_CLIENT_ATTRIB_BITS

Table 6.2: Attribute groups

Type code	Explanation
В	Boolean
C	Color (floating-point R, G, B, and A values)
CI	Color index (floating-point index value)
T	Texture coordinates (floating-point s, t, r, q
	values)
N	Normal coordinates (floating-point x, y, z val-
	ues)
V	Vertex, including associated data
Z	Integer
Z^+	Non-negative integer
Z_k, Z_{k*}	k-valued integer (k * indicates k is minimum)
R	Floating-point number
R^+	Non-negative floating-point number
$R^{[a,b]}$	Floating-point number in the range $[a, b]$
R^k	k-tuple of floating-point numbers
P	Position $(x, y, z, w \text{ floating-point coordinates})$
D	Direction $(x, y, z \text{ floating-point coordinates})$
M^4	4×4 floating-point matrix
I	Image
A	Attribute stack entry, including mask
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is mini-
	mum)

Table 6.3:State variable types

to the selected light is returned; with textures, where only the selected texture or texture parameter is returned; and with evaluator maps, where only the selected map is returned. Finally, a "-" in the attribute column indicates that the indicated value is not included in any attribute group (and thus can not be pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**).

The M and m entries for initial minmax table values represent the maximum and minimum possible representable values, respectively.

6.2 State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of **GetBooleanv**, **GetIntegerv**, **GetFloatv**, or **GetDoublev** are listed with just one of these commands – the one that is most appropriate given the type of the data to be returned. These state variables cannot be obtained using **IsEnabled**. However, state variables for which **IsEnabled** is listed as the query command can also be obtained using **GetBooleanv**, **GetIntegerv**, **GetFloatv**, and **GetDoublev**. State variables for which any other command is listed as the query command can be obtained only by using that command.

State table entries which are required only by the imaging subset (see section 3.6.2) are typeset against a gray background.

Attribute	1		I		I		1		1	I		I		ļ								1		I		
Sec.	2.6.1		2.6.1		2.6.1		2.6.1		3.4	2.6.1		2.6.1		2.6.1			2.6.1			2.6.1		2.6.1		2.6.1		
Description	When $\neq 0$, indicates	begin/end object	Previous vertex in	Begin/End line	Indicates if <i>line-vertex</i>	is the first	First vertex of a	Begin/End line loop	Line stipple counter	Vertices inside of	Begin/End polygon	Number of	polygon-vertices	Previous two vertices	in a Begin/End	triangle strip	Number of vertices so	far in triangle strip: 0,	1, or more	Triangle strip A/B	vertex pointer	Vertices of the quad	under construction	Number of vertices so	far in quad strip: 0, 1,	2, or more
Initial Value	0		I		1		1		-	1		1		-			-			-		-		1		
Get Cmnd	1		I		I		I		Ι	I		I		I			Ι			I		Ι		I		
Type	Z_{11}		Λ		B		Λ		+Z	A imes u		+Z		$2 \times V$			Z^3			Z^2		$3 \times V$		Z_4		
Get value	I		Ι		I		1		Ι	Ι		Ι		I			Ι			Ι		Ι		Ι		

Table 6.4. GL Internal begin-end state variables (inaccessible)

Attribute	current	current	current		current	-				ļ			current	current	current		current		current			current		current
Sec.	2.7	2.7	2.7		2.7	2.6		2.6		2.6			2.12	2.12	2.12		2.12		2.12			2.12		2.6.2
Description	Current color	Current color index	Current texture	coordinates	Current normal	Color associated with	last vertex	Color index associated	with last vertex	Texture coordinates	associated with last	vertex	Current raster position	Current raster distance	Color associated with	raster position	Color index associated	with raster position	Texture coordinates	associated with raster	position	Raster position valid	bit	Edge flag
Initial Value	1, 1, 1, 1	1	0,0,0,1		0, 0, 1	I		I		I			0,0,0,1	0	1, 1, 1, 1		1		0,0,0,1			True		True
Get Cmnd	GetIntegerv, GetFloatv	GetIntegerv, GetFloatv	GetFloatv		GetFloatv	I		I		I			GetFloatv	GetFloatv	GetIntegerv, GetFloatv		GetIntegerv, GetFloatv		GetFloatv			GetBooleanv		GetBooleanv
Type	C	CI	T		N	С		CI		L			R^4	R^+	C		CI		T			B		B
Get value	CURRENT_COLOR	CURRENT_INDEX	CURRENT_TEXTURE_COORDS		CURRENT_NORMAL	I		-		1			CURRENT_RASTER_POSITION	CURRENT_RASTER_DISTANCE	CURRENT_RASTER_COLOR		CURRENT_RASTER_INDEX		CURRENT_RASTER_TEXTURE_COORDS			CURRENT_RASTER_POSITION_VALID		EDGEFLAG

Table 6.5. Current Values and Associated Data

Sec. Attribute	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2.8 vertex-array	2 8 vertex-arrav	ł
Description	Vertex array enable	Coordinates per vertex	Type of vertex coordinates	Stride between vertices	Pointer to the vertex array	Normal array enable	Type of normal coordinates	Stride between normals	Pointer to the normal array	Color array enable	Colors per vertex	Type of color components	Stride between colors	Pointer to the color array	Index array enable	Type of indices	Stride between indices	Pointer to the index array	Texture coordinate array enable	Coordinates per element	Type of texture coordinates	Stride between texture coordinates	Pointer to the texture coordinate	Edoe flao arrav enable	Ctuido hotmoon odeo Ace
Initial Value	False	4	FLOAT	0	0	False	FLOAT	0	0	False	4	FLOAT	0	0	False	FLOAT	0	0	False	4	FLOAT	0	0	False	
Get Cmnd	IsEnabled	GetIntegerv	GetIntegerv	GetIntegerv	GetPointerv	IsEnabled	GetIntegerv	GetIntegerv	GetPointerv	IsEnabled	GetIntegerv	GetIntegerv	GetIntegerv	GetPointerv	IsEnabled	GetIntegerv	GetIntegerv	GetPointerv	IsEnabled	GetIntegerv	GetIntegerv	GetIntegerv	GetPointerv	TsF,nahled	Cot Internet
Type	В	Z^+	Z_4	$^{+Z}$	Y	В	Z_5	Z^+	${}^{\Lambda}$	В	Z^+	Z_8	Z^+	Y	B	Z_4	Z^+	Y	B	$^{+Z}$	Z_4	$^{+Z}$	\overline{A}	В	+2
Get value	VERTEX_ARRAY	VERTEX_ARRAY_SIZE	VERTEX_ARRAY_TYPE	VERTEX_ARAAY_STRIDE	VERTEX_ARRAY_POINTER	NORMAL_ARRAY	NORMAL_ARRAY_TYPE	NORMAL_ARRAY_STRIDE	NORMAL_ARRAY_POINTER	COLOR_ARRAY	COLOR_ARRAY_SIZE	COLOR_ARRAY_TYPE	COLOR_ARRAY_STRIDE	COLOR_ARRAY_POINTER	INDEX_ARAY	INDEX_ARRAY_TYPE	INDEX_ARRAY_STRIDE	INDEX_ARRAY_POINTER	TEXTURE_COORD_ARRAY	TEXTURE_COORD_ARRAY_SIZE	TEXTURE_COORD_ARRAY_TYPE	TEXTURE_COORD_ARRAY_STRIDE	TEXTURE_COORD_ARRAY_POINTER	RDGE ELAC ABBAV	

Table 6.6. Vertex Array Data

Attribute	I	l	l	I	viewport	viewport	1	1		l	$\operatorname{transform}$	transform/enable	transform/enable	transform	transform/enable
Sec.	3.6.3	2.10.2	2.10.2	2.10.2	2.10.1	2.10.1	3.6.3	2.10.2	2.10.2	2.10.2	2.10.2	2.10.3	2.10.3	2.11	2.11
Description	Color matrix stack	Model-view matrix stack	Projection matrix stack	Texture matrix stack	Viewport origin $\&$ extent	Depth range near $\&$ far	Color matrix stack pointer	Model-view matrix stack pointer	Projection matrix stack pointer	Texture matrix stack pointer	Current matrix mode	Current normal normalization on/off	Current normal rescaling on/off	User clipping plane coefficients	<i>i</i> th user clipping plane enabled
Initial Value	Identity	Identity	Identity	Identity	see 2.10.1	0,1	1	Ţ	1	1	MODELVIEW	False	False	0,0,0,0	False
Get Cmnd	GetFloatv	GetFloatv	GetFloatv	$\operatorname{GetFloatv}$	GetIntegerv	GetFloatv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	IsEnabled	IsEnabled	GetClipPlane	IsEnabled
Type	$2 * imes M^4$	$32 * \times M^4$	$2 * imes M^4$	$2 * imes M^4$	$4 \times Z$	$2 imes R^+$	+2	z^+	+Z	+Z	Z_4	В	B	$6 * imes R^4$	6* imes B
Get value	COLOR_MATRIX	MODELVIEW_MATRIX	PROJECTION_MATRIX	TEXTURE_MATRIX	VIEWPORT	DEPTH_RANGE	COLOR_MATRIX_STACK_DEPTH	MODELVIEW_STACK_DEPTH	PROJECTION_STACK_DEPTH	TEXTURE_STACK_DEPTH	MATRIX_MODE	NORMALIZE	RESCALE_NORMAL	CLIP_PLANEi	CLIP_PLANE <i>i</i>

Table 6.7. Transformation state

-

		1	1						
Attribute	fog	$_{\rm gof}$	goj		goj	goj	goj	fog/enable	lighting
Sec.	3.10	3.10	3.10		3.10	3.10	3.10	3.10	2.13.7
Description	Fog color	Fog index	Exponential fog	$\operatorname{density}$	Linear fog start	Linear fog end	Fog mode	True if fog enabled	ShadeModel setting
Initial Value	0,0,0,0	0	1.0		0.0	1.0	EXP	False	RIDOTH
Get Cmnd	GetFloatv	GetFloatv	GetFloatv		GetFloatv	GetFloatv	GetIntegerv	IsEnabled	GetIntegerv
Type	С	CI	R		R	R	Z_3	B	Z^+
Get value	FOG-COLOR	FOG_INDEX	FOG_DENSITY		FOG_START	FOG_END	FOG_MODE	FOG	SHADE_MODEL

Table 6.8. Coloring

Attribute	lighting/enable	lighting/enable	lighting	lighting	lighting	lighting	lighting	lighting	lighting	lighting	lighting	lighting	lighting
Sec.	2.13.1	2.13.3	2.13.3	2.13.3	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1
Description	True if lighting is enabled	True if color tracking is enabled	Material properties tracking current color	Face(s) affected by color tracking	Ambient material color	Diffuse material color	Specular material color	Emissive mat. color	Specular exponent of material	Ambient scene color	Viewer is local	Use two-sided lighting	Color control
Initial Value	False	False	AMBIENT_AND_DIFFUSE	FRONT AND BACK	(0.2, 0.2, 0.2, 1.0)	(0.8, 0.8, 0.8, 1.0)	(0.0, 0.0, 0.0, 1.0)	(0.0, 0.0, 0.0, 1.0)	0.0	(0.2, 0.2, 0.2, 1.0)	False	False	SINGLE_COLOR
Get Cmnd	IsEnabled	IsEnabled	GetIntegerv	GetIntegerv	GetMaterialfv	GetMaterialfv	GetMaterialfv	GetMaterialfv	GetMaterialfv	GetFloatv	GetBooleanv	GetBooleanv	GetIntegerv
Type	B	В	Z_5	Z_3	2 imes C	2 imes C	2 imes C	2 imes C	2 imes R	Э	B	В	Z_2
			COLOR_MATERIAL_PARAMETER	COLOR_MATERIAL_FACE						LIGHT_MODEL_AMBIENT	LIGHT_MODEL_LOCAL_VIEWER	LIGHT_MODEL_TWO_SIDE	LIGHT MODEL COLOR CONTROL

6.2. STATE TABLES

Table 6.9. Lighting (see also Table 2.7 for defaults)

Attribute	lighting	lighting	lighting	lighting	lighting	lighting	lighting		lighting	lighting	lighting	lighting/enable	lighting
Sec.	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1	2.13.1		2.13.1	2.13.1	2.13.1	2.13.1	2.13.1
Description	Ambient intensity of light i	Diffuse intensity of light i	Specular intensity of light i	Position of light i	Constant atten. factor	Linear atten. factor	Quadratic atten.	factor	Spotlight direction of light i	Spotlight exponent of light i	Spot. angle of light i	True if light i enabled	$a_m, d_m, and s_m$ for color index lighting
Initial Value	(0.0,0.0,0.0,1.0)	see 2.5	see 2.5	(0.0,0.0,1.0,0.0)	1.0	0.0	0.0		(0.0,0.0,-1.0)	0.0	180.0	False	0,1,1
Get Cmnd	GetLightfv	GetLightfv	GetLightfv	GetLightfv	GetLightfv	GetLightfv	$\operatorname{GetLightfv}$		GetLightfv	GetLightfv	GetLightfv	IsEnabled	GetMaterialfv
Type	$8 * \times C$	$8 * \times C$	$8 * \times C$	8 * imes P	$8 * \times R^+$	$8 * \times R^+$	$8 * \times R^+$		8* imes D	$8 * \times R^+$	$8 * \times R^+$	8 * imes B	2 imes 3 imes R
Get value	AMBIBNT	DIFFUSE	SPECULAR	NOITISOT	CONSTANT_ATTENUATION	LINEAR_ATTENUATION	QUADRATIC_ATTENUATION		SPOT_DIRECTION	SPOT_EXPONENT	SPOT_CUTOFF	LIGHT_i	COLOR_INDEXES

Table 6.10. Lighting (cont.)

Version 1.2.1 - April 1, 1999

Sec. Attribute	3.3 point	3.3 point/enable	3.4 line	3.4 line/enable	3.4.2 line	3.4.2 line	3.4.2 line/enable	3.5.1 polygon/enable	3.5.1 polygon		3.5.1 polygon	3.5 polygon/enable		3.5.4 polygon	3.5.5 polygon	3.5.5 polygon	3.5.5 polygon/enable		3.5.5 polygon/enable		3.5.5 polygon/enable		3.5 polygon-stipple	359 nolvon /anabla
Description	Point size	Point antialiasing on	Line width	Line antialiasing on	Line stipple 5	Line stipple repeat 5		Polygon culling enabled	it/back facing		Polygon trontface CW/CCW indicator	Polygon antialiasing	on	Polygon rasterization 3 mode (front & back)	Polygon offset factor 5		nable	tor POLNT mode rasterization	Polygon offset enable	rasterization	Polygon offset enable 5 for FILL mode	rasterization	Polygon stipple	ahaha
Initial Value	1.0	False	1.0	False	1's		False	False	BACK	1	CCW	False		FILL	0	0	False		False		False		1's	$F_{\alpha I_{\alpha \alpha}}$
Get Cmnd	GetFloatv	IsEnabled	GetFloatv	IsEnabled	GetIntegerv	GetIntegerv	IsEnabled	IsEnabled	GetIntegerv		GetIntegerv	IsEnabled		GetIntegerv	GetFloatv	GetFloatv	IsEnabled		IsEnabled		IsEnabled		GetPolygonStipple	Is Hinchlod
Type	R^+	В	R^+	В	Z^+	Z^+	В	В	Z_3	t	Z_2	В		$2 imes Z_3$	R	R	В		В		В		Ι	Ч
Get value	POINT_SIZE	POINT_SMOOTH	LINE_WIDTH	LINE_SMOOTH	LINE_STIPPLE_PATTERN	LINE_STIPPLE_REPEAT	LINE_STIPPLE	CULLFACE	CULL_FACE_MODE		FRONT_FACE	POLYGON_SMOOTH		POLYGON_MODE	POLYGON-OFFSET_FACTOR	POLYGON_OFFSET_UNITS	POLYGON_OFFSET_POINT		POLYGON_OFFSET_LINE		POLYGON-OFFSET_FILL		1	DOLVCON STIDDLE

Table 6.11. Rasterization

:. Attribute	10 texture/enable	.8 texture	-	·	·	·	-	-	·	-	-	-	·	-
Sec.	3.8.10	3.8.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8	3.8
Description	True if x D texturing is enabled; x is 1, 2, or 3	Texture object bound to TEXTURE_xD	xD texture image at l.o.d. i	xD texture image i 's specified width	2D texture image i's specified height	3D texture image i's specified depth	xD texture image i 's specified border width	xD texture image i 's internal image format	xD texture image i 's red resolution	xD texture image i 's green resolution	xD texture image i 's blue resolution	xD texture image i 's alpha resolution	xD texture image i 's luminance resolution	xD texture image i 's
Initial Value	False	0	see 3.8	0	0	0	0	1	0	0	0	0	0	0
Get Cmnd	IsEnabled	GetIntegerv	GetTexImage	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter	GetTexLevelParameter
Type	3 imes B	$3 \times Z^+$	I imes u	$n \times Z^+$	$n \times Z^+$	$n \times Z^+$	$u imes Z^+$	$n imes Z_{42}$	$n \times Z^+$	$^+Z imes u$	$u imes Z^+$	$n \times Z^+$	$n \times Z^+$	$n imes Z^+$
Get value	TEXTURE_2D	TEXTURE_BINDING_&D	TEXTURE_CD	TEXTURE_WIDTH	TEXTURE_HEIGHT	TEXTURE_DEPTH	TEXTURE_BORDER	TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS)	TEXTURE_RED_SIZE	TEXTURE_GREEN_SIZE	TEXTURE.BLUE.SIZE	TEXTURE_ALPHA_SIZE	TEXTURELUMINANCE_SIZE	TEXTURE_INTENSITY_SIZE

Table 6.12. Texture Objects

Version 1.2.1 - April 1, 1999

TEXTURE DOLDER_COLOR $2^+ \times C$ Get TexParameter $0,0,0,0$ Texture border color3.8textuTEXTURE MIN_FULTER $2^+ \times Z_6$ Get TexParametersee 3.8Texture minification3.8.5textuTEXTURE MAG_FULTER $2^+ \times Z_2$ Get TexParametersee 3.8Texture magnification3.8.5textuTEXTURE MAG_FULTER $2^+ \times Z_2$ Get TexParametersee 3.8Texture magnification3.8.6textuTEXTURE MAG_FULTER $2^+ \times Z_3$ Get TexParameterREPEATTexture wrap mode S3.8.textuTEXTURE WRAP-R $2^+ \times Z_3$ Get TexParameterREPEATTexture wrap mode S3.8.textuTEXTURE WRAP-R $1^+ \times Z_3$ Get TexParameterREPEATTexture wrap mode S3.8.textuTEXTURE MAPLAR $1^+ \times Z_3$ Get TexParameterREPEATTexture wrap mode S3.8.textuTEXTURE MAPLAR $1^+ \times Z_3$ Get TexParameterREPEATTexture wrap mode S3.8.textuTEXTURE MANLOD $1^+ \times Z_3$ Get TexParameterRef 3.8.Texture wrap mode S3.8.textuTEXTURE MANLOD $n \times R$ Get TexParametersee 3.8.8Texture wrap mode S3.8.textuTEXTURE MANLOD $n \times R$ Get TexParametersee 3.8.8Texture wrap mode S3.8.textuTEXTURE MANLOD $n \times R$ Get TexParametersee 3.8.8Texture wrap mode S3.8.textuTEXTURE MANLOD $n \times R$ Get TexParametersee 3.8.8Tex	Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
BIL $2^+ \times Z_6$ GetTexParametersee 3.8Texture minification3.8.5BIL $2^+ \times Z_2$ GetTexParametersee 3.8Texture magnification3.8.6BIL $3^+ \times Z_3$ GetTexParametersee 3.8Texture magnification3.8.6 $3^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode S3.8 $1^+ \times Z_3$ GetTexParameterN1Texture object priority3.8.8 $1^- \times R_3$ GetTexParameterivsee 3.8.8Texture residency3.8.8 $1^- \times R_3$ 1000 Maximun level of3.8 $1^- N = N = N = N = N = N = N = N = N = N $	TEXTURE BORDER COLOR	$2^+ imes C$	GetTexParameter	0,0,0,0	Texture border color	3.8	texture
BR $2^+ \times Z_2$ GetTexParametersee 3.8function3.8.6 $2R$ $2^+ \times Z_3$ GetTexParametersee 3.8Texture magnification3.8.6 $3^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode S3.8 $2^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode S3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode S3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode R3.8 $1^+ \times Z_3$ GetTexParameter N1Texture object priority3.8 r $2^+ \times B^{[0,1]}$ GetTexParameter N1000Minimum level of3.8 r $n \times R$ GetTexParameter N-1000Minimum level of3.8 $n \times R$ GetTexParameter N-1000Maximum level of3.8 $n \times R$ GetTexParameter N-1000Maximum level of3.8 $n \times R$ $n \times R$ GetTexParameter N-1000Maximum level of <td< td=""><td>TEXTURE_MIN_FILTER</td><td>$2^+ imes Z_6$</td><td>GetTexParameter</td><td>see 3.8</td><td>Texture minification</td><td>3.8.5</td><td>texture</td></td<>	TEXTURE_MIN_FILTER	$2^+ imes Z_6$	GetTexParameter	see 3.8	Texture minification	3.8.5	texture
TER $2^+ \times Z_2$ GetTexParametersee 3.8Texture magnification3.8.6 $3^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode S3.8 $2^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode T3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode T3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode T3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode R3.8 $1^+ \times Z_3$ GetTexParameterfy1Texture object priority3.8.8 $1^- \times Z_3$ GetTexParameterfy1000Minimum level of3.8 $n \times R$ GetTexParameterfy1000Maximum level of3.8 $n \times R$ $n \times R$ GetTexParameterfy1000Maximum level of3.8 $n \times R$ $n \times R$ GetTexParameterfy1000Maximum level of3.8 $n \times R$ $n \times R$ GetTexParameterfy $n \otimes N$ $n \otimes N$ $n \otimes N$ $n \times R$ $n \times R$ GetTexParameterfy $n \otimes N$ $n \otimes N$ n					function		
Image: constraint of the state of the st	TEXTURE_MAG_FILTER	$2^+ \times Z_2$	GetTexParameter	see 3.8	Texture magnification	3.8.6	texture
$3^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode S3.8 $2^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode T3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode T3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode R3.8 $2^+ \times R^{[0,1]}$ GetTexParameterfy1Texture object priority3.8.8 r $2^+ \times R$ GetTexParameterfy1000Minimum level of3.8.8 r $n \times R$ GetTexParameterfy-1000Minimum level of3.8 $n \times R$ GetTexParameterfy1000Maximum level3.8					function		
$2^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode T3.8 $1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode R3.8 $2^+ \times R^{[0,1]}$ GetTexParameterfy1Texture object priority3.8.8r $2^+ \times R$ GetTexParameterfy1000Minimum level of3.8.8r $n \times R$ GetTexParameterfy-1000Minimum level of3.8.8n $n \times R$ GetTexParameterfy-1000Minimum level of3.8n $n \times R$ GetTexParameterfy-1000Minimum level of3.8n $n \times R$ GetTexParameterfy0Maximum level of3.8m $n \times R$ GetTexParameterfy1000Maximum level of3.8m $n \times R$ GetTexParameterfy0Base texture array3.8m $n \times R$ GetTexParameterfy0Maximum level of3.8m $n \times R$ GetTexParameterfy1000Maximum level of3.8m $n \times R$ GetTexParameterfy0Base texture array3.8m $n \times R$ GetTexParameterfy1000Maximum texture3.8	TEXTURE_WRAP_S	$3^+ \times Z_3$	GetTexParameter	REPEAT	Texture wrap mode S	3.8	texture
$1^+ \times Z_3$ GetTexParameterREPEATTexture wrap mode R3.8 $2^+ \times R^{[0,1]}$ GetTexParameterfy1Texture object priority3.8.8 n $2^+ \times B$ GetTexParameterivsee 3.8.8Texture residency3.8.8 $n \times R$ GetTexParameterfv-1000Minimum level of3.8.8 $n \times R$ GetTexParameterfv-1000Minimum level of3.8.8 $n \times R$ GetTexParameterfv1000Maximum level of3.8	TEXTURE_WRAP_T	$2^+ imes Z_3$	GetTexParameter	REPEAT	Texture wrap mode T	3.8	texture
$2^+ \times R^{[0,1]}$ GetTexParameterfy 1 Texture object priority 3.8.8 r $2^+ \times B$ GetTexParameterix see 3.8.8 Texture residency 3.8.8 n $n \times R$ GetTexParameterix see 3.8.8 Texture residency 3.8.8 n $n \times R$ GetTexParameterix -1000 Minimum level of detail 3.8.8 m $n \times R$ GetTexParameterix 1000 Maximum level of detail 3.8 m $n \times R$ GetTexParameterix 1000 Maximum level of detail 3.8 m $n \times R$ GetTexParameterix 1000 Maximum level of detail 3.8 m $n \times R$ GetTexParameterix 1000 Maximum level of detail 3.8 m $n \times R$ GetTexParameterix 0 Base texture array 3.8 m $n \times R$ GetTexParameterix 1000 Maximum texture 3.8 m $n \times R$ GetTexParameterix 1000 Maximum texture 3.8	TEXTURE_WRAP_R	$1^+ imes Z_3$	GetTexParameter	REPEAT	Texture wrap mode R	3.8	texture
r $2^+ \times B$ GetTexParameterivsee 3.8.8Texture residency3.8.8 $n \times R$ GetTexParameteriv-1000Minimum level of3.8 $n \times R$ GetTexParameteriv1000Maximum level of3.8 $n \times R$ GetTexParameteriv1000Maximum level of3.8 $m \times R$ GetTexParameteriv000Maximum level of3.8 $m \times R$ GetTexParameteriv1000Maximum level of3.8 $m \times R$ GetTexParameteriv00Base texture array3.8 $m \times R$ GetTexParameteriv1000Maximum texture3.8 $m \times R$ GetTexParameteriv0Base texture array3.8 $m \times R$ GetTexParameteriv1000Maximum texture3.8	TEXTURE PRIORITY	$2^+ imes R^{[0,1]}$	GetTexParameterfv	1	Texture object priority	3.8.8	texture
$ \begin{array}{ c c c c c c c c c c c c c c c c c c c$	TEXTURE_RESIDENT	$2^+ imes B$	GetTexParameteriv	see 3.8.8	Texture residency	3.8.8	texture
$n \times R$ GetTexParameterfy 1000 Maximum level of 3.8 The second se	TEXTUREMIN-LOD	n imes R	GetTexParameterfv	-1000	Minimum level of	3.8	texture
$n \times R$ GetTexParameterfy 1000 Maximum level of 3.8 BL $n \times R$ GetTexParameterfy 0 Base texture array 3.8 BL $n \times R$ GetTexParameterfy 00 Maximum texture 3.8 BL $n \times R$ GetTexParameterfy 1000 Maximum texture 3.8					detail		
$n \times R$ GetTexParameterfy0Base texture array3.8 $n \times R$ GetTexParameterfy1000Maximum texture3.8 $n \times R$ GetTexParameterfy1000Maximum texture3.8	TEXTUREMAX_LOD	n imes R	GetTexParameterfv	1000	Maximum level of	3.8	texture
$n \times R$ GetTexParameterfv0Base texture array3.8 $n \times R$ GetTexParameterfv1000Maximum texture3.8 $n \times R$ GetTexParameterfvannotationannotationannotation					detail		
$n \times R$ GetTexParameterfv 1000 Maximum texture 3.8 array level	TEXTUREBASELEVEL	n imes R	GetTexParameterfv	0	Base texture array	3.8	texture
array level	TEXTUREMAX_LEVEL	n imes R	${f Get Tex Parameter fv}$	1000	Maximum texture	3.8	texture
					array level		

Table 6.13. Texture Objects (cont.)

Attribute	texture	texture	texture/enable	texture	texture	texture
Sec.	3.8.9	3.8.9	2.10.4	2.10.4	2.10.4	2.10.4
Description	Texture application function	Texture environment color	Texgen enabled $(x is S, T, R, \text{ or } Q)$	Texgen plane equation coefficients (for S, T, R, and Q)	Texgen object linear coefficients (for S, T, R, and Q)	Function used for texgen (for S, T, R, and Q
Initial Value	MODULATE	0,0,0,0	False	see 2.10.4	see 2.10.4	EYE_LINEAR
Get Cmnd	GetTexEnviv	GetTexEnvfv	IsEnabled	GetTexGenfv	GetTexGenfv	GetTexGeniv
Type	Z_4	\mathcal{D}	4 imes B	$4 imes R^4$	$4 imes R^4$	$4 \times Z_3$
Get value	TEXTURE_ENV_MODE	TEXTURE_ENV_COLOR	TEXTURE_GEN_x	EYE_PLANE	OBJECT_PLANE	TEXTURE GEN MODE

Table 6.14. Texture Environment and Generation

CHAPTER 6. STATE AND STATE REQUESTS

-

г

- 1

-

Attribute	scissor/enable	scissor	color-buffer/enable	color-buffer	color-buffer		stencil-buffer/enable	stencil-buffer	stencil-buffer	stencil-buffer	stencil-buffer	stencil-buffer		stencil-buffer		depth-buffer/enable	depth-buffer		color-buffer/enable	color-buffer		color-buffer		color-buffer	color-buffer	color-buffer/enable	color-buffer/enable	color-buffer/enable	color-buffer
Sec.	4.1.2	4.1.2	4.1.3	4.1.3	4.1.3		4.1.4	4.1.4	4.1.4	4.1.4	4.1.4	4.1.4		4.1.4		4.1.5	4.1.5		4.1.6	4.1.6		4.1.6		4.1.6	4.1.6	4.1.7	4.1.8	4.1.8	4.1.8
Description	Scissoring enabled	Scissor box	Alpha test enabled	Alpha test function	Alpha test reference	value	Stenciling enabled	Stencil function	Stencil mask	Stencil reference value	Stencil fail action	Stencil depth buffer	fail action	Stencil depth buffer	pass action	Depth buffer enabled	Depth buffer test	function	Blending enabled	Blending source	function	Blending destination	function	Blending equation	Constant blend color	Dithering enabled	Index logic op enabled	Color logic op enabled	Logic op function
Initial Value	False	see $4.1.2$	False	ALWAYS	0		False	ALWAYS	1's	0	KEEP	KEEP		ИЕЕР		False	LESS		False	ONE		ZERO		FUNC_ADD	0,0,0,0	True	False	False	СОРҮ
Get Cmnd	IsEnabled	GetIntegerv	IsEnabled	GetIntegerv	GetIntegerv		IsEnabled	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv		GetIntegerv		IsEnabled	GetIntegerv		IsEnabled	GetIntegerv		GetIntegerv		GetIntegerv	$\operatorname{Get} Floatv$	IsEnabled	IsEnabled	IsEnabled	GetIntegerv
Type	В	$4 \times Z$	B	Z_8	R^+		B	Z_8	Z^+	Z^+	Z_6	Z_6		Z_6		B	Z_8		B	Z_{13}		Z_{12}		Z_5	C	B	B	В	Z_{16}
Get value	SCISSOR_TEST	SCISSOR_BOX	ALPHA_TEST	ALPHA_TEST_FUNC	ALPHA_TEST_REF		STENCIL_TEST	STENCIL_FUNC	STENCIL_VALUE_MASK	STENCIL_REF	STENCILFAIL	STENCIL-PASS_DEPTH_FAIL		STENCIL_PASS_DEPTH_PASS		DEPTH_TEST	DEPTH_FUNC		BLEND	BLEND_SRC		BLEND_DST		BLEND_EQUATION	BLEND_COLOR	DITHER	INDEX_LOGIC_OP (v1.0: GL_LOGIC_OP)	COLOR-LOGIC-OP	LOGIC-OP-MODE

Table 6.15. Pixel Operations

Attribute	color-buffer	color-buffer	color-buffer	depth-buffer	stencil-buffer	color-buffer	color-buffer	depth-buffer	stencil-buffer	accum-buffer
Sec.	4.2.1	4.2.2	4.2.2	4.2.2	4.2.2	4.2.3	4.2.3	4.2.3	4.2.3	4.2.3
Description	Buffers selected for drawing	Color index writemask	Color write enables; R, G, B, or A	Depth buffer enabled for writing	Stencil buffer writemask	Color buffer clear value (RGBA mode)	Color buffer clear value (color index mode)	Depth buffer clear value	Stencil clear value	Accumulation buffer clear value
Initial Value	see 4.2.1	1's	on_{J}	True	$1'_S$	0'0'0'0	0	1	0	0
Get Cmnd	GetIntegerv	GetIntegerv	GetBooleanv	GetBooleanv	GetIntegerv	GetFloatv	GetFloatv	GetIntegerv	GetIntegerv	GetFloatv
Type	Z_{10*}	Z^+	4 imes B	В	+Z	\mathcal{D}	IJ	R^+	Z^+	$4 \times R^+$
Get value	DRAW_BUFFER	INDEX_WRITEMASK	COLOR_WRITEMASK	DEPTH_WRITEMASK	STENCIL-WRITEMASK	COLOR-CLEAR-VALUE	INDEX_CLEAR_VALUE	DEPTH_CLEAR_VALUE	STENCIL-CLEAR-VALUE	ACCUM-CLEAR-VALUE

Table 6.16. Framebuffer Control

Attribute	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store	pixel-store
Sec.	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3	4.3
Description	Value of UNPACK_SWAP_BYTES	Value of UNPACK_LSB_FIRST	Value of UNPACK_IMAGE_HEIGHT	Value of UNPACK_SKIP_IMAGES	Value of UNPACK_ROW_LENGTH	Value of UNPACK_SKIP_ROWS	Value of UNPACK_SKIP_PIXELS	Value of UNPACK_ALIGNMENT	Value of PACK_SWAP_BYTES	Value of PACK_LSB_FIRST	Value of PACK_IMAGE_HEIGHT	Value of PACK_SKIP_IMAGES	Value of PACK_ROW_LENGTH	Value of PACK_SKIP_ROWS	Value of PACK_SKIP_PIXELS	Value of PACK_ALIGNMENT
Initial Value	False	False	0	0	0	0	0	4	False	False	0	0	0	0	0	4
Get Cmnd	GetBooleanv	GetBooleanv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	GetBooleanv	GetBooleanv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv
Type	В	В	Z^+	Z^+	Z^+	Z^+	Z^+	Z^+	В	В	Z^+	^{+}Z	Z^+	Z^+	Z^+	^+Z
Get value	UNPACK_SWAP_BYTES	UNPACK_LSB_FIRST	UNPACK_IMAGE_HEIGHT	UNPACK_SKIP_IMAGES	UNPACK_ROW_LENGTH	UNPACK_SKIP_ROWS	UNPACK_SKIP_PIXBLS	UNPACK_ALIGNMENT	PACK_SWAP_BYTES	PACK_LSB_FIRST	PACK_IMAGE_HEIGHT	PACK_SKIP_IMAGES	PACK_ROW_LENGTH	PACK_SKIP_ROWS	PACK_SKIP_PIXELS	PACK_ALIGNMENT

Table 6.17. Pixels

Attribute	pixel	pixel	pixel	pixel	pixel		pixel		pixel/enable		pixel/enable			pixel/enable			I										pixel		pixel	
Sec.	4.3	4.3	4.3	4.3	4.3		4.3		3.6.3		3.6.3			3.6.3			3.6.3	3.6.3		3.6.3		3.6.3					3.6.3		3.6.3	
Description	True if colors are mapped	True if stencil values are mapped	Value of INDEX_SHIFT	Value of INDEX_OFFSET	Value of x -SCALE; x is	RED, GREEN, BLUE, ALPHA, or DEPTH	Value of x -BIAS; x is	one of RED, GREEN, BLUE, ALPHA, or DEPTH	True if color table	lookup is done	True if post	convolution color table	lookup is done	True if post color	matrix color table	Iookup 15 done	Color tables	Color tables' internal	image format	Color tables' specified	width	Color table component	resolution; x is RED,	GREEN, BLUE, ALPHA,	LUMINANCE, or	INTENSITY	Scale factors applied	to color table entries	Bias factors applied to color table entries	
Initial Value	False	False	0	0	1		0		False		False			False			empty	RGBA		0		0					1, 1, 1, 1		0,0,0,0	
Get Cmnd	GetBooleanv	GetBooleanv	GetIntegerv	GetIntegerv	GetFloatv		GetFloatv		IsEnabled		IsEnabled			IsEnabled			GetColorTable	GetColorTable-	Parameteriv	GetColorTable-	Parameteriv	GetColorTable-	Parameteriv				GetColorTable-	Parameterfv	GetColorTable- Parameterfv	
Type	B	В	Ζ	Ζ	R		R		В		В			В			3 imes I	$2 imes 3 imes Z_{42}$		$2 imes 3 imes Z^+$		$6 imes 2 imes 3 imes Z^+$					$3 imes R^4$		$3 imes R^4$	
Get value	MAP_COLOR	MAP_STENCIL	INDEX_SHIFT	INDEX_OFFSET	x-SCALE		<i>x</i> -BIAS		COLOR-TABLE		POST_CONVOLUTION_COLOR_TABLE			POST_COLOR_MATRIX_COLOR_TABLE			COLOR-TABLE	COLOR-TABLE-FORMAT		COLOR_TABLE_WIDTH		COLOR_TABLE_x_SIZE					COLOR-TABLE-SCALE		COLOR-TABLE-BIAS	

CHAPTER 6. STATE AND STATE REQUESTS

Attribute	pixel/enable	pixel/enable	pixel/enable	I	I	pixel	pixel	pixel	pixel	1	I	1
Sec.	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3	4.3	4.3	3.6.3	3.6.3	4.3	4.3	4.3
Description	True if 1D convolution is done	True if 2D convolution is done	True if separable 2D convolution is done	Convolution filters	Separable convolution filter	Convolution border color	Convolution border mode	Scale factors applied to convolution filter entries	Bias factors applied to convolution filter entries	Convolution filter internal format	Convolution filter width	Convolution filter height
Initial Value	False	False	False	empty	empty	0,0,0,0	REDUCE	1, 1, 1, 1	0,0,0,0	RGBA	0	0
Get Cmnd	IsEnabled	IsEnabled	IsEnabled	GetConvolution- Filter	GetSeparable- Filter	GetConvolution- Parameterfy	GetConvolution- Parameteriv	GetConvolution- Parameterfy	GetConvolution- Parameterfy	GetConvolution- Parameteriv	GetConvolution- Parameteriv	GetConvolution- Parameteriv
Type	В	В	В	2 imes I	2 imes I	3 imes C	$3 \times Z_4$	$3 imes R^4$	$3 imes R^4$	$3 imes Z_{42}$	$3 \times Z^+$	$2 imes Z^+$
						RDER-COLOR						

Table 6.19. Pixels (cont.)

Attribute	pixel	pixel	pixel	pixel	pixel/enable	I	1	1	1	1
Sec.	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3
Description	Component scale factors after convolution: x is RED, GREEN, BLUE, or ALPHA	Component bias factors after convolution: x is RED, GREEN, BLUE, or ALPHA	Component scale factors after color matrix; x is RED, GREEN, BLUE, or ALPHA	Component bias factors after color matrix; x is RED, GREEN, BLUE, or ALPHA	True if histogramming is enabled	Histogram table	Histogram table width	Histogram table internal format	Histogram table component resolution; x is RED, GREEN, BLUE, ALPHA, or LUMINANCE	True if histogramming consumes pixel groups
Initial Value	1	0	1	0	False	empty	0	RGBA	0	False
Get Cmnd	GetFloatv	GetFloatv	GetFloatv	GetFloatv	IsEnabled	GetHistogram	GetHistogram- Parameteriv	GetHistogram- Parameteriv	GetHistogram- Parameteriv	GetHistogram- Parameteriv
Type	R	R	R	R	В	Ι	$2 imes Z^+$	$2 imes Z_{42}$	$5 \times 2 \times Z^+$	В
Get value	POST_CONVOLUTION_2_SCALE	POST-CONVOLUTION-z-BIAS	POST_COLOR_MATRIX_z_SCALE	POST-COLOR-MATRIX_2-BIAS	HISTOGRAM	HISTOGRAM	HISTOGRAM_WIDTH	HISTOGRAM_FORMAT	HISTOGRAMzSIZE	HISTOGRAM_SINK

Table 6.20. Pixels (cont.)

Attribute	pixel/enable	1	I	1		pixel	pixel	1				1				1	pixel
Sec.	3.6.3	3.6.3	3.6.3	3.6.3)	4.3	4.3	4.3				4.3				4.3	4.3
Description	True if minmax is enabled	Minmax table	Minmax table internal	True if minmax	consumes pixel groups	x zoom factor	y zoom factor	RGBA PixelMap	translation tables; x is	a map name from	Table 3.3	Index PixelMap	translation tables; x is	a map name from	Table 3.3	Size of table x	Read source buffer
Initial Value	False	(M,M,M,M),(m,m,m,m)	RGBA	False		1.0	1.0	0's				0's				1	see 4.3.2
Get Cmnd	IsEnabled	GetMinmax	GetMinmax-	r arameteriv Get Minmax-	Parameteriv	GetFloatv	GetFloatv	GetPixelMap				GetPixelMap				GetIntegerv	GetIntegerv
Type	В	R^n	Z_{42}	В	I	R	R	$8 \times 32 * \times R$				$2 \times 32 * \times Z$				Z^+	Z_3
Get value	MINMAX	MINMAX	MINMAX_FORMAT	MINMAX SINK		ZOOM	ZOOM_Y	x				ĸ				xSIZE	READ_BUFFER

Table 6.21. Pixels (cont.)

Version 1.2.1 - April 1, 1999

Attribute		ļ	ļ	1	1	1	eval/enable		eval/enable		eval	eval	eval	eval	eval/enable		
Sec.	5.1	5.1	5.1	5.1	5.1	5.1	5.1		5.1		5.1	5.1	5.1	5.1	5.1		
Description	1d map order	2d map orders	1d control points	2d control points	1d domain endpoints	2d domain endpoints	1d map enables: x is	map type	2d map enables: x is	map type	1d grid endpoints	2d grid endpoints	1d grid divisions	2d grid divisions	True if automatic	normal generation	enabled
Initial Value	1	1,1	see 5.1	see 5.1	see 5.1	see 5.1	False		False		0,1	0,1;0,1	1	1,1	False		
Get Cmnd	$\operatorname{GetMapiv}$	$\operatorname{GetMapiv}$	$\operatorname{GetMapfv}$	$\operatorname{GetMapfv}$	$\operatorname{GetMapfv}$	$\operatorname{GetMapfv}$	IsEnabled		IsEnabled		GetFloatv	GetFloatv	GetFloatv	GetFloatv	IsEnabled		
Type	$9 \times Z_{8*}$	$9 imes 2 imes Z_{8*}$	$9 imes 8* imes R^n$	$9 \times 8 * \times 8 * \times R^n$	9 imes 2 imes R	9 imes 4 imes R	9 imes B		$9 \times B$		2 imes R	4 imes R	Z^+	$2 imes Z^+$	В		
		r	ſ	r		r	Ē		-					F	-	-	

Table 6.22. Evaluators (GetMap takes a map name)

Soc Attributo	AUDUINUA	hint		hint	hint	hint	hint	
сос С		5.6		5.6	5.6	5.6	5.6	
Docemination	TIOTINTITICE	Z ₃ GetIntegerv DONT_CARE Perspective correction	hint	GetIntegerv DONT_CARE Point smooth hint	GetIntegerv DONT_CARE Line smooth hint	GetIntegerv DONT_CARE Polygon smooth hint	Fog hint	
Initial Value	vauue	DONT_CARE		DONT_CARE	DONT_CARE	DONT_CARE	DONT_CARE	
Get		GetIntegerv		GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv DONT_CARE Fog hint	
Tuno	турс	Z_3		Z_3	Z_3	Z_3	Z_3	
	OCE VALUE	PERSPECTIVE_CORRECTION_HINT		POINT_SMOOTH_HINT	LUIH HTOOM2 SMOOTH HINT	POLYGON SMOOTH HINT	FOG_HINT	

Table 6.23. Hints

Table 6 24	Implementation	Dependent	Values
10.010 0.21.	in promotion of the	Dopondone	101000

Version 1.2.1 - April 1, 1999

CHAPTER 6. STATE AND STATE REQUESTS

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.	Attribute
	+Z	GetIntegerv	0	Number of auxiliary buffers	4.2.1	
	В	GetBooleanv	Ι	True if color buffers store rgba	2.7	
	В	GetBooleanv	I	True if color buffers store indexes	2.7	
DOUBLEBUFFER	В	GetBooleanv	I	True if front & back buffers exist	4.2.1	I
	В	GetBooleanv	I	True if left & right buffers exist	9	I
ALIASED_POINT_SIZE_RANGE	$2 imes R^+$	GetFloatv	1,1	Range (lo to hi) of aliased point sizes	3.3	I
SMOOTH_POINT_SIZE_RANGE (v1.1: POINT_SIZE_RANGE)	$2 imes R^+$	GetFloatv	1,1	Range (lo to hi) of antialiased point sizes	3.3	1
SMOOTH_POINT_SIZE_GRANULARITY (vl.1: POINT_SIZE_GRANULARITY)	R^+	GetFloatv	I	Antialiased point size granularity	3.3	I
ALIASED_LINE_WIDTH_RANGE	$2 imes R^+$	GetFloatv	1,1	Range (lo to hi) of aliased line widths	3.4	
SMOOTH_LINE_WIDTH_RANGE (v1.1: LINE_WIDTH_RANGE)	$2 imes R^+$	GetFloatv	1,1	Range (lo to hi) of antialiased line widths	3.4	I
SMOOTH LINE WIDTH GRANULARITY (vi 1: LINE WIDTH GRANULARITY)	R^+	GetFloatv	I	Antialiased line width granularity	3.4	I
MAX_CONVOLUTION_WIDTH	$3 imes Z^+$	GetConvolution- Parameteriv	က	Maximum width of convolution filter	4.3	
MAX_CONVOLUTION_HEIGHT	$2 imes Z^+$	GetConvolution- Parameteriv	3	Maximum height of convolution filter	4.3	1
MAX_ELEMENTS_INDICES	Z^+	GetIntegerv	1	Recommended maximum number of Draw Range Ele- ments indices	2.8	1
MAX_BLEMENTS_VERTICES	Z^+	GetIntegerv	Ι	Recommended maximum number of DrawRangeEle- ments vertices	2.8	1

6.2. STATE TABLES

Table 6.25. More Implementation Dependent Values

Sec. Attribute		1	I	
Sec.	4	4	4	4
Description	Number of bits in x color buffer component; x is one of RED, GREEN, BLUE, ALPHA, OT INDEX	Number of depth buffer planes	Number of stencil planes	Number of bits in x accumulation buffer component (x is RED, GREEN, BLUE, or ALPHA
Initial Value	1	I	I	1
Get Cmnd	GetIntegerv	GetIntegerv	GetIntegerv	GetIntegerv
Type	+Z	Z^+	z^+	Z^+
Get value	STIELx	DEPTH_BITS	STENCIL_BITS	ACCUM & BITS

CHAPTER 6. STATE AND STATE REQUESTS

Table 6.26. Implementation Dependent Pixel Depths

Attribute	list							-	1							select		select	feedback		feedback	feedback	-	I	
Sec.	5.4	5.4			5.4			9	9		9	9		5.2	5.2	5.2		5.2	5.3		5.3	5.3	2.5	5.5	
Description	Setting of ListBase	number of display list	under construction; 0	if none	Mode of display list	under construction;	undefined if none	Server attribute stack	Server attribute stack	pointer	Client attribute stack	Client attribute stack	pointer	Name stack depth	RenderMode setting	Selection buffer	pointer	Selection buffer size	Feedback buffer	pointer	Feedback buffer size	Feedback type	Current error code(s)	True if there is a	corresponding error
Initial Value	0	0			0			empty	0		empty	0		0	RENDER	0		0	0		0	2D	0	False	
Get Cmnd	GetIntegerv	GetIntegerv			GetIntegerv			—	GetIntegerv		—	GetIntegerv		GetIntegerv	GetIntegerv	GetPointerv		GetIntegerv	GetPointerv		GetIntegerv	GetIntegerv	$\operatorname{Get}\operatorname{Error}$	Η	
Type	Z^+	Z^+			^+Z			$16 * \times A$	+Z		$16 * \times A$	^+Z		^+Z	Z_3	Λ		Z^+	Λ		Z^+	Z_5	$n imes Z_8$	n imes B	
Get value	TISIA TSIA	LIST_INDEX			TIST_MODE				ATTRIB_STACK_DEPTH		I	CLIENT_ATTRIB_STACK_DEPTH		NAME STACK DEPTH	HENDER MODE	SELECTION_BUFFER_POINTER		SELECTION_BUFFER_SIZE	FEEDBACK_BUFFER_POINTER		FEEDBACK_BUFFER_SIZE	FEEDBACK_BUFFER_TYPE		1	

6.2. STATE TABLES

-

Table 6.27. Miscellaneous

٦

Т Т

Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, highperformance implementations, and in many software vendors choosing to port to OpenGL.

A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

Rule 1 For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.

Rule 2 Changes to the following state values have no side effects (the use of any other state value is not affected by the change):

Required:

- Framebuffer contents (all bitplanes)
- The color buffers enabled for writing

- The values of matrices other than the top-of-stack matrices
- Scissor parameters (other than enable)
- Writemasks (color, index, depth, stencil)
- Clear values (color, index, depth, stencil, accumulation)
- Current values (color, index, normal, texture coords, edgeflag)
- Current raster color, index and texture coordinates.
- Material properties (ambient, diffuse, specular, emission, shininess)

Strongly suggested:

- Matrix mode
- Matrix stack depths
- Alpha test parameters (other than enable)
- Stencil parameters (other than enable)
- Depth test parameters (other than enable)
- Blend parameters (other than enable)
- Logical operation parameters (other than enable)
- Pixel storage and transfer state
- Evaluator state (except as it affects the vertex data generated by the evaluators)
- Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)

Corollary 1 Fragment generation is invariant with respect to the state values marked with • in Rule 2.

Corollary 2 The window coordinates (x, y, and z) of generated fragments are also invariant with respect to

Required:

- Current values (color, color index, normal, texture coords, edge-flag)
- Current raster color, color index, and texture coordinates
- Material properties (ambient, diffuse, specular, emission, shininess)

Rule 3 The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the alpha test, for instance, are the alpha test enable, the alpha test function, and the alpha test reference value).

Corollary 3 Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.

A.4 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating point values may be represented using different formats in different renderers (hardware and software), many OpenGL state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

- 1. The CURRENT_RASTER_TEXTURE_COORDS must be maintained correctly at all times, including periods while texture mapping is not enabled, and when the GL is in color index mode.
- 2. When requested, texture coordinates returned in feedback mode are always valid, including periods while texture mapping is not enabled, and when the GL is in color index mode.
- 3. The error semantics of upward compatible OpenGL revisions may change. Otherwise, only additions can be made to upward compatible revisions.
- 4. GL query commands are not required to satisfy the semantics of the **Flush** or the **Finish** commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.
- 5. Application specified point size and line width must be returned as specified when queried. Implementation dependent clamping affects the values only while they are in use.
- 6. Bitmaps and pixel transfers do not cause selection hits.
- 7. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on

the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the stencil comparison function; it limits the effect of the update of the stencil buffer.

- 8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is FLAT, all of the points or lines generated by a single polygon will have the same color.
- 9. A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in COMPILE mode, errors should not be generated while the list is being created.
- 10. **RasterPos** does not change the current raster index from its default value in an RGBA mode GL context. Likewise, **RasterPos** does not change the current raster color from its default value in a color index GL context. Both the current raster index and the current raster color can be queried, however, regardless of the color mode of the GL context.
- 11. A material property that is attached to the current color via Color-Material always takes the value of the current color. Attempts to change that material property via Material calls have no effect.
- 12. **Material** and **ColorMaterial** can be used to modify the RGBA material properties, even in a color index context. Likewise, **Material** can be used to modify the color index material properties, even in an RGBA context.
- 13. There is no atomicity requirement for OpenGL rendering commands, even at the fragment level.
- 14. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in FILL mode, and the fragments generated by the rasterization of "narrow" polygons may not form a continuous array.
- 15. OpenGL does not force left- or right-handedness on any of its coordinates systems. Consider, however, the following conditions: (1) the object coordinate system is right-handed; (2) the only commands used to manipulate the model-view matrix are **Scale** (with positive scaling values only), **Rotate**, and **Translate**; (3) exactly one of either **Frustum** or **Ortho** is used to set the projection matrix; (4) the near value

is less than the far value for **DepthRange**. If these conditions are all satisfied, then the eye coordinate system is right-handed and the clip, normalized device, and window coordinate systems are left-handed.

- 16. ColorMaterial has no effect on color index lighting.
- 17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge (that is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon, and the coordinates of vertex A (resp. B) are identical to those of vertex C (resp. D), and the state of the the coordinate transfomations is identical when A, B, C, and D are specified). Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.
- 18. OpenGL state continues to be modified in FEEDBACK mode and in SELECT mode. The contents of the framebuffer are not modified.
- 19. The current raster position, the user defined clip planes, the spot directions and the light positions for LIGHT*i*, and the eye planes for texgen are transformed when they are specified. They are not transformed during a **PopAttrib**, or when copying a context.
- 20. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.
- 21. For any GL and framebuffer state, and for any group of GL commands and arguments, the resulting GL and framebuffer state is identical whether the GL commands and arguments are executed normally or from a display list.

Appendix C

Version 1.1

OpenGL version 1.1 is the first revision since the original version 1.0 was released on 1 July 1992. Version 1.1 is upward compatible with version 1.0, meaning that any program that runs with a 1.0 GL implementation will also run unchanged with a 1.1 GL implementation. Several additions were made to the GL, especially to the texture mapping capabilities, but also to the geometry and fragment operations. Following are brief descriptions of each addition.

C.1 Vertex Array

Arrays of vertex data may be transferred to the GL with many fewer commands than were previously necessary. Six arrays are defined, one each storing vertex positions, normal coordinates, colors, color indices, texture coordinates, and edge flags. The arrays may be specified and enabled independently, or one of the pre-defined configurations may be selected with a single command.

The primary goal was to decrease the number of subroutine calls required to transfer non-display listed geometry data to the GL. A secondary goal was to improve the efficiency of the transfer; especially to allow direct memory access (DMA) hardware to be used to effect the transfer. The additions match those of the EXT_vertex_array extension, except that static array data are not supported (because they complicated the interface, and were not being used), and the pre-defined configurations are added (both to reduce subroutine count even further, and to allow for efficient transfer of array data).

C.2 Polygon Offset

Depth values of fragments generated by the rasterization of a polygon may be shifted toward or away from the origin, as an affine function of the window coordinate depth slope of the polygon. Shifted depth values allow coplanar geometry, especially facet outlines, to be rendered without depth buffer artifacts. They may also be used by future shadow generation algorithms.

The additions match those of the EXT_polygon_offset extension, with two exceptions. First, the offset is enabled separately for POINT, LINE, and FILL rasterization modes, all sharing a single affine function definition. (Shifting the depth values of the outline fragments, instead of the fill fragments, allows the contents of the depth buffer to be maintained correctly.) Second, the offset bias is specified in units of depth buffer resolution, rather than in the [0,1] depth range.

C.3 Logical Operation

Fragments generated by RGBA rendering may be merged into the framebuffer using a logical operation, just as color index fragments are in GL version 1.0. Blending is disabled during such operation because it is rarely desired, because many systems could not support it, and to match the semantics of the EXT_blend_logic_op extension, on which this addition is loosely based.

C.4 Texture Image Formats

Stored texture arrays have a format, known as the *internal format*, rather than a simple count of components. The internal format is represented as a single enumerated value, indicating both the organization of the image data (LUMINANCE, RGB, etc.) and the number of bits of storage for each image component. Clients can use the internal format specification to suggest the desired storage precision of texture images. New *base formats*, ALPHA and INTENSITY, provide new texture environment operations. These additions match those of a subset of the EXT_texture extension.

C.5 Texture Replace Environment

A common use of texture mapping is to replace the color values of generated fragments with texture color data. This could be specified only indirectly

in GL version 1.0, which required that client specified "white" geometry be modulated by a texture. GL version 1.1 allows such replacement to be specified explicitly, possibly improving performance. These additions match those of a subset of the EXT_texture extension.

C.6 Texture Proxies

Texture proxies allow a GL implementation to advertise different maximum texture image sizes as a function of some other texture parameters, especially of the internal image format. Clients may use the proxy query mechanism to tailor their use of texture resources at run time. The proxy interface is designed to allow such queries without adding new routines to the GL interface. These additions match those of a subset of the EXT_texture extension, except that implementations return allocation information consistent with support for complete mipmap arrays.

C.7 Copy Texture and Subtexture

Texture array data can be specified from framebuffer memory, as well as from client memory, and rectangular subregions of texture arrays can be redefined either from client or framebuffer memory. These additions match those defined by the EXT_copy_texture and EXT_subtexture extensions.

C.8 Texture Objects

A set of texture arrays and their related texture state can be treated as a single object. Such treatment allows for greater implementation efficiency when multiple arrays are used. In conjunction with the subtexture capability, it also allows clients to make gradual changes to existing texture arrays, rather than completely redefining them. These additions match those of the EXT_texture_object extension, with slight additions to the texture residency semantics.

C.9 Other Changes

1. Color indices may now be specified as unsigned bytes.

- 2. Texture coordinates s, t, and r are divided by q during the rasterization of points, pixel rectangles, and bitmaps. This division was documented only for lines and polygons in the 1.0 version.
- 3. The line rasterization algorithm was changed so that vertical lines on pixel borders rasterize correctly.
- 4. Separate pixel transfer discussions in chapter 3 and chapter 4 were combined into a single discussion in chapter 3.
- 5. Texture alpha values are returned as 1.0 if there is no alpha channel in the texture array. This behavior was unspecified in the 1.0 version, and was incorrectly documented in the reference manual.
- 6. Fog start and end values may now be negative.
- 7. Evaluated color values direct the evaluation of the lighting equation if **ColorMaterial** is enabled.

C.10 Acknowledgements

OpenGL 1.1 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Kurt Akeley, Silicon Graphics Bill Armstrong, Evans & Sutherland Andy Bigos, 3Dlabs Pat Brown, IBM Jim Cobb, Evans & Sutherland Dick Coulter, Digital Equipment Bruce D'Amora, GE Medical Systems John Dennis, Digital Equipment Fred Fisher, Accel Graphics Chris Frazier, Silicon Graphics Todd Frazier, Evans & Sutherland Tim Freese, NCD Ken Garnett, NCD Mike Heck, Template Graphics Software Dave Higgins, IBM Phil Huxley, 3Dlabs

Dale Kirkland, Intergraph Hock San Lee, Microsoft Kevin LeFebvre, Hewlett Packard Jim Miller, IBM Tim Misner, SunSoft Jeremy Morris, 3Dlabs Israel Pinkas, Intel Bimal Poddar, IBM Lyle Ramshaw, Digital Equipment Randi Rost, Hewlett Packard John Schimpf, Silicon Graphics Mark Segal, Silicon Graphics Igor Sinyak, Intel Jeff Stevenson, Hewlett Packard Bill Sweeney, SunSoft Kelvin Thompson, Portable Graphics Neil Trevett, 3Dlabs Linas Vepstas, IBM Andy Vesper, Digital Equipment Henri Warren, Megatek Paula Womack, Silicon Graphics Mason Woo, Silicon Graphics Steve Wright, Microsoft

Appendix D

Version 1.2

OpenGL version 1.2, released on March 16, 1998, is the second revision since the original version 1.0. Version 1.2 is upward compatible with version 1.1, meaning that any program that runs with a 1.1 GL implementation will also run unchanged with a 1.2 GL implementation.

Several additions were made to the GL, especially to texture mapping capabilities and the pixel processing pipeline. Following are brief descriptions of each addition.

D.1 Three-Dimensional Texturing

Three-dimensional textures can be defined and used. In-memory formats for three-dimensional images, and pixel storage modes to support them, are also defined. The additions match those of the EXT_texture3D extension.

One important application of three-dimensional textures is rendering volumes of image data.

D.2 BGRA Pixel Formats

BGRA extends the list of host-memory color formats. Specifically, it provides a component order matching file and framebuffer formats common on Windows platforms. The additions match those of the **EXT_bgra** extension.

D.3 Packed Pixel Formats

Packed pixels in host memory are represented entirely by one unsigned byte, one unsigned short, or one unsigned integer. The fields with the packed pixel

are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes and their unpacking counterparts all work correctly with packed pixels.

The additions match those of the EXT_packed_pixels extension, with the further addition of reversed component order packed formats.

D.4 Normal Rescaling

Normals may be rescaled by a constant factor derived from the modelview matrix. Rescaling can operate faster than renormalization in many cases, while resulting in the same unit normals.

The additions are based on the EXT_rescale_normal extension.

D.5 Separate Specular Color

Lighting calculations are modified to produce a primary color consisting of emissive, ambient and diffuse terms of the usual GL lighting equation, and a secondary color consisting of the specular term. Only the primary color is modified by the texture environment; the secondary color is added to the result of texturing to produce a single post-texturing color. This allows highlights whose color is based on the light source creating them, rather than surface properties.

The additions match those of the EXT_separate_specular_color extension.

D.6 Texture Coordinate Edge Clamping

GL normally clamps such that the texture coordinates are limited to exactly the range [0, 1]. When a texture coordinate is clamped using this algorithm, the texture sampling filter straddles the edge of the texture image, taking half its sample values from within the texture image, and the other half from the texture border. It is sometimes desirable to clamp a texture without requiring a border, and without using the constant border color.

A new texture clamping algorithm, CLAMP_TO_EDGE, clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. The color returned when clamping is derived only from texels at the edge of the texture image.

The additions match those of the SGIS_texture_edge_clamp extension.

D.7 Texture Level of Detail Control

Two constraints related to the texture level of detail parameter λ are added. One constraint clamps λ to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. Image array specification is necessarily integral, rather than continuous. By providing separate, continuous clamping of the λ parameter, it is possible to avoid "popping" artifacts when higher resolution images are provided.

The additions match those of the SGIS_texture_lod extension.

D.8 Vertex Array Draw Element Range

A new form of **DrawElements** that provides explicit information on the range of vertices referred to by the index set is added. Implementations can take advantage of this additional information to process vertex data without having to scan the index data to determine which vertices are referenced.

The additions match those of the EXT_draw_range_elements extension.

D.9 Imaging Subset

The remaining new features are primarily intended for advanced image processing applications, and may not be present in all GL implementations. The are collectively referred to as the *imaging subset*.

D.9.1 Color Tables

A new RGBA-format color lookup mechanism is defined in the pixel transfer process, providing additional lookup capabilities beyond the existing lookup. The key difference is that the new lookup tables are treated as one-dimensional images with internal formats, like texture images and convolution filter images. Thus the new tables can operate on a subset of the components of passing pixel groups. For example, a table with internal format ALPHA modifies only the A component of each pixel group, leaving the R, G, and B components unmodified. Three independent lookups may be performed: prior to convolution; after convolution and prior to color matrix transformation; after color matrix transformation and prior to gathering pipeline statistics.

Methods to initialize the color lookup tables from the framebuffer, in addition to the standard memory source mechanisms, are provided.

Portions of a color lookup table may be redefined without reinitializing the entire table. The affected portions may be specified either from host memory or from the framebuffer.

The additions match those of the EXT_color_table and EXT_color_subtable extensions.

D.9.2 Convolution

One- or two-dimensional convolution operations are executed following the first color table lookup in the pixel transfer process. The convolution kernels are themselves treated as one- and two-dimensional images, which can be loaded from application memory or from the framebuffer.

The convolution framework is designed to accommodate threedimensional convolution, but that API is left for a future extension.

The additions match those of the EXT_convolution and HP_convolution_border_modes extensions.

D.9.3 Color Matrix

A 4x4 matrix transformation and associated matrix stack are added to the pixel transfer path. The matrix operates on RGBA pixel groups, using the equation

where

C' = MC,

$$C = \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

and M is the 4×4 matrix on the top of the color matrix stack. After the matrix multiplication, each resulting color component is scaled and biased by a programmed amount. Color matrix multiplication follows convolution.

The color matrix can be used to reassign and duplicate color components. It can also be used to implement simple color space conversions.

The additions match those of the SGI_color_matrix extension.

D.9.4 Pixel Pipeline Statistics

Pixel operations that count occurences of specific color component values (histogram) and that track the minimum and maximum color component values (minmax) are performed at the end of the pixel transfer pipeline. An optional mode allows pixel data to be discarded after the histogram and/or minmax operations are completed. Otherwise the pixel data continues on to the next operation unaffected.

The additions match those of the EXT_histogram extension.

D.9.5 Constant Blend Color

A constant color that can be used to define blend weighting factors may be defined. A typical usage is blending two RGB images. Without the constant blend factor, one image must have an alpha channel with each pixel set to the desired blend factor.

The additions match those of the EXT_blend_color extension.

D.9.6 New Blending Equations

Blending equations other than the normal weighted sum of source and destination components may be used.

Two of the new equations produce the minimum (or maximum) color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

The other two equations are similar to the default blending equation, but produce the difference of its left and right hand sides, rather than the sum. Image differences are useful in many image processing applications.

The additions match those of the EXT_blend_minmax and EXT_blend_subtract extensions.

D.10 Acknowledgements

OpenGL 1.2 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Kurt Akeley, Silicon Graphics Bill Armstrong, Evans & Sutherland Otto Berkes, Microsoft

Pierre-Luc Bisaillon, Matrox Graphics Drew Bliss, Microsoft David Blythe, Silicon Graphics Jon Brewster, Hewlett Packard Dan Brokenshire, IBM Pat Brown, IBM Newton Cheung, S3 Bill Clifford, Digital Jim Cobb, Parametric Technology Bruce D'Amora, IBM Kevin Dallas, Microsoft Mahesh Dandapani, Rendition Daniel Daum, AccelGraphics Suzy Deffeyes, IBM Peter Doyle, Intel Jay Duluk, Raycer Craig Dunwoody, Silicon Graphics Dave Erb, IBM Fred Fisher, AccelGraphics / Dynamic Pictures Celeste Fowler, Silicon Graphics Allen Gallotta, ATI Ken Garnett, NCD Michael Gold, Nvidia / Silicon Graphics Craig Groeschel, Metro Link Jan Hardenbergh, Mitsubishi Electric Mike Heck, Template Graphics Software Dick Hessel, Raycer Graphics Paul Ho, Silicon Graphics Shawn Hopwood, Silicon Graphics Jim Hurley, Intel Phil Huxley, 3Dlabs Dick Jay, Template Graphics Software Paul Jensen, 3Dfx Brett Johnson, Hewlett Packard Michael Jones, Silicon Graphics Tim Kelley, Real3D Jon Khazam, Intel Louis Khouw, Sun Dale Kirkland, Intergraph Chris Kitrick, Raycer

Don Kuo, S3 Herb Kuta, Quantum 3D Phil Lacroute, Silicon Graphics Prakash Ladia, S3 Jon Leech, Silicon Graphics Kevin Lefebvre, Hewlett Packard David Ligon, Raycer Graphics Kent Lin, S3 Dan McCabe, S3 Jack Middleton, Sun Tim Misner, Intel Bill Mitchell, National Institute of Standards Jeremy Morris, 3Dlabs Gene Munce, Intel William Newhall, Real3D Matthew Papakipos, Nvidia / Raycer Garry Paxinos, Metro Link Hanspeter Pfister, Mitsubishi Electric Richard Pimentel, Parametric Technology Bimal Poddar, IBM / Intel Rob Putney, IBM Mike Quinlan, Real3D Nate Robins, University of Utah Detlef Roettger, Elsa Randi Rost, Hewlett Packard Kevin Rushforth, Sun Richard S. Wright, Real3D Hock San Lee, Microsoft John Schimpf, Silicon Graphics Stefan Seeboth, ELSA Mark Segal, Silicon Graphics Bob Seitsinger, S3 Min-Zhi Shao, S3 Colin Sharp, Rendition Igor Sinyak, Intel Bill Sweeney, Sun William Sweeney, Sun Nathan Tuck, Raycer Doug Twillenger, Sun John Tynefeld, 3dfx

Kartik Venkataraman, Intel Andy Vesper, Digital Equipment Henri Warren, Digital Equipment / Megatek Paula Womack, Silicon Graphics Steve Wright, Microsoft David Yu, Silicon Graphics Randy Zhao, S3

Appendix E

Version 1.2.1

OpenGL version 1.2.1, released on October 14, 1998, introduced ARB extensions (see Appendix F). The only ARB extension defined in this version is multitexture, allowing application of multiple textures to a fragment in one rendering pass. Multitexture is based on the SGIS_multitexture extension, simplified by removing the ability to route texture coordinate sets to arbitrary texture units.

A new corollary discussing display list and immediate mode invariance was added to Appendix B on April 1, 1999.

Appendix F

ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review Board (ARB) are described in this chapter. These extensions are not required to be supported by a conformant OpenGL implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

In order not to compromise the readability of the core specification, ARB extensions are not integrated into the core language; instead, they are presented in this chapter, as changes to the core.

F.1 Naming Conventions

To distinguish ARB extensions from core OpenGL features and from vendorspecific extensions, the following naming conventions are used:

- A unique *name string* of the form "GL_ARB_*name*" is associated with each extension. If the extension is supported by an implementation, this string will be present in the EXTENSIONS string described in section 6.1.11.
- All functions defined by the extension will have names of the form *Function***ARB**
- All enumerants defined by the extension will have names of the form $\ensuremath{\textit{NAME}_ARB}.$

F.2 Multitexture

Multitexture adds support for multiple texture units. The capabilities of the multiple texture units are identical, except that evaluation and feedback are supported only for texture unit 0. Each texture unit has its own state vector which includes texture vertex array specification, texture image and filtering parameters, and texture environment application.

The texture environments of the texture units are applied in a pipelined fashion whereby the output of one texture environment is used as the input fragment color for the next texture environment. Changes to texture client state and texture server state are each routed through one of two selectors which control which instance of texture state is affected.

The specification is written using four texture units though the actual number supported is implementation dependent and can be larger or smaller than four.

The name string for multitexture is GL_ARB_multitexture.

F.2.1 Dependencies

Multitexture requires features of OpenGL 1.1.

F.2.2 Issues

The extension currently requires a separate texture coordinate input for each texture unit. Modification to allow routing and/or broadcasting texcoords and **TexGen** output would be useful, possibly as a future extension layered on multitexture.

F.2.3 Changes to Section 2.6 (Begin/End Paradigm)

Amend paragraphs 2 and 3

Each vertex is specified with two, three, or four coordinates. In addition, a *current normal*, multiple *current texture coordinate sets*, and *current color* may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how a texture image is mapped onto a primitive. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture units supported is implementation dependent but must be at least one. The number of active textures supported can be queried with the state MAX_TEXTURE_UNITS_ARB. Primary and secondary colors are associated with each vertex (see section 3.9). These *associated* colors are either based on the current color or produced by lighting, depending on whether or not lighting is enabled. Texture coordinates are similarly associated with each vertex. Multiple sets of texture coordinates may be associated with a vertex. Figure F.1 summarizes the association of auxiliary data with a transformed vertex to produce a *processed vertex*.

Amend paragraph 6

Before colors have been assigned to a vertex, the state required by a vertex is the vertex's coordinates, the current normal, the current edge flag (see section 2.6.2), the current material properties (see section 2.13.2), and the multiple current texture coordinate sets. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex's coordinates, its edge flag, its assigned colors, and its multiple texture coordinate sets.

F.2.4 Changes to Section 2.7 (Vertex Specification)

Amend paragraph 2

Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

```
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( T coords );
```

specify the current homogeneous texture coordinates, named s, t, r, and q. The **TexCoord1** family of commands set the s coordinate to the provided single argument while setting t and r to 0 and q to 1. Similarly, **TexCoord2** sets s and t to the specified values, r to 0 and q to 1; **TexCoord3** sets s, t, and r, with q set to 1, and **TexCoord4** sets all four texture coordinates.

Implementations may support more than one texture unit, and thus more than one set of texture coordinates. The commands

take the coordinate set to be modified as the *texture* parameter. *texture* is a symbolic constant of the form **TEXTURE***i*_**ARB**, indicating that texture coordinate set *i* is to be modified. The constants obey **TEXTURE***i*_**ARB** =

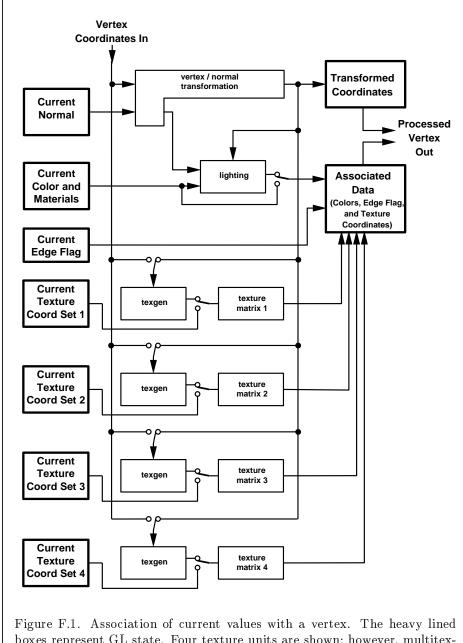


Figure F.1. Association of current values with a vertex. The heavy lined boxes represent GL state. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. **TEXTUREO_ARB** + i (i is in the range 0 to k-1, where k is the implementationdependent number of texture units defined by MAX_TEXTURE_UNITS_ARB).

The **TexCoord** commands are exactly equivalent to the corresponding **MultiTexCoordARB** commands with *texture* set to **TEXTUREO_ARB**.

Gets of CURRENT_TEXTURE_COORDS return the texture coordinate set defined by the value of ACTIVE_TEXTURE_ARB.

Specifying an invalid texture coordinate set for the *texture* argument of **MultiTexCoordARB** results in undefined behavior.

F.2.5 Changes to Section 2.8 (Vertex Arrays)

Amend paragraph 1

The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to 5 plus the value of MAX_TEXTURE_UNITS_ARB arrays: one each to store vertex coordinates, edge flags, colors, color indices, normals, and one or more texture coordinate sets. The commands ...

Insert between paragraph 2 and 3

In implementations which support more than one texture unit, the command

void ClientActiveTextureARB(enum texture);

is used to select the vertex array client state parameters to be modified by the **TexCoordPointer** command and the array affected by **EnableClientState** and **DisableClientState** with parameter **TEXTURE_COORD_ARRAY**. This command sets the client state variable **CLIENT_ACTIVE_TEXTURE_ARB**. Each texture unit has a client state vector which is selected when this command is invoked. This state vector includes the vertex array state. This call also selects which texture units' client state vector is used for queries of client state.

Specifying an invalid *texture* generates the error INVALID_ENUM. Valid values of *texture* are the same as for the **MultiTexCoordARB** commands described in section 2.7.

Amend final paragraph

If the number of supported texture units (the value of MAX_TEXTURE_UNITS_ARB) is k, then the client state required to implement vertex arrays consists of 5 + k boolean values, 5 + k memory pointers, 5 + k integer stride values, 4 + k symbolic constants representing array types, and 3 + k integers representing values per element. In the initial state, the boolean values are each disabled, the memory pointers are each null, the strides are each zero, the array types are each FLOAT, and the integers representing values per element are each four.

F.2.6 Changes to Section 2.10.2 (Matrices)

Amend paragraph 8

For each texture unit, a 4×4 matrix is applied to the corresponding texture coordinates. This matrix is applied as

m_1	m_5	m_9	m_{13}	$\langle s \rangle$
m_2	m_6	m_{10}	m_{14}	t
m_3	m_7	m_{11}	m_{15}	r ,
$\setminus m_4$	m_8	m_{12}	m_{16} /	$\backslash q$ /

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to **TEXTURE** causes the already described matrix operations to apply to the texture matrix.

There is also a corresponding texture matrix stack for each texture unit. To change the stack affected by matrix operations, set the *active texture unit selector* by calling

void ActiveTextureARB(enum texture);

The selector also affects calls modifying texture environment state, texture coordinate generation state, texture binding state, and queries of all these state values as well as current texture coordinates and current raster texture coordinates.

Specifying an invalid *texture* generates the error INVALID_ENUM. Valid values of *texture* are the same as for the **MultiTexCoordARB** commands described in section 2.7.

The active texture unit selector may be queried by calling **GetIntegerv** with *pname* set to ACTIVE_TEXTURE_ARB.

There is a stack of matrices for each of matrix modes MODELVIEW, PROJECTION, and COLOR, and for each texture unit. For MODELVIEW mode, the stack depth is at least 32 (that is, there is a stack of at least 32 modelview matrices). For the other modes, the depth is at least 2. Texture matrix stacks for all texture units have the same depth. The current matrix in any mode is the matrix on the top of the stack for that mode.

```
void PushMatrix( void );
```

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

```
void PopMatrix( void );
```

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error STACK_UNDERFLOW; pushing a matrix onto a full stack generates STACK_OVERFLOW.

When the current matrix mode is **TEXTURE**, the texture matrix stack of the active texture unit is pushed or popped.

The state required to implement transformations consists of a fourvalued integer indicating the current matrix mode, one stack of at least two 4×4 matrices for each of COLOR, PROJECTION, each texture unit, TEXTURE, and a stack of at least $32 \ 4 \times 4$ matrices for MODELVIEW. Each matrix stack has an associated stack pointer. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial matrix mode is MODELVIEW. The initial value of ACTIVE_TEXTURE_ARB is TEXTUREO_ARB.

F.2.7 Changes to Section 2.10.4 (Generating Texture Coordinates)

Amend paragraph 4

The state required for texture coordinate generation for each texture unit comprises a three-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR. The initial state has the texture generation function disabled for all texture coordinates. The initial values of p_i for s are all 0 except p_1 which is one; for t all the p_i are zero except p_2 , which is 1. The values of p_i for r and q are all 0. These values of p_i apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

For implementations which support more than one texture unit, there is texture coordinate generation state for each unit. The texture coordinate generation state which is affected by the **TexGen**, **Enable**, and **Disable** operations is set with **ActiveTextureARB**.

F.2.8 Changes to Section 2.12 (Current Raster Position)

Amend paragraph 2

The state required for the current raster position consists of three window coordinates x_w , y_w , and z_w , a clip coordinate w_c value, an eye coordinate distance, a valid bit, and associated data consisting of a color and multiple texture coordinate sets. It is set using one of the **RasterPos** commands:

void RasterPos{234}{sifd}(T coords); void RasterPos{234}{sifd}v(T coords);

RasterPos4 takes four values indicating x, y, z, and w. **RasterPos3** (or **RasterPos2**) is analogous, but sets only x, y, and z with w implicitly set to 1 (or only x and y with z implicitly set to 0 and w implicitly set to 1).

Gets of CURRENT_RASTER_TEXTURE_COORDS are affected by the setting of the state ACTIVE_TEXTURE_ARB.

Modify figure 2.7

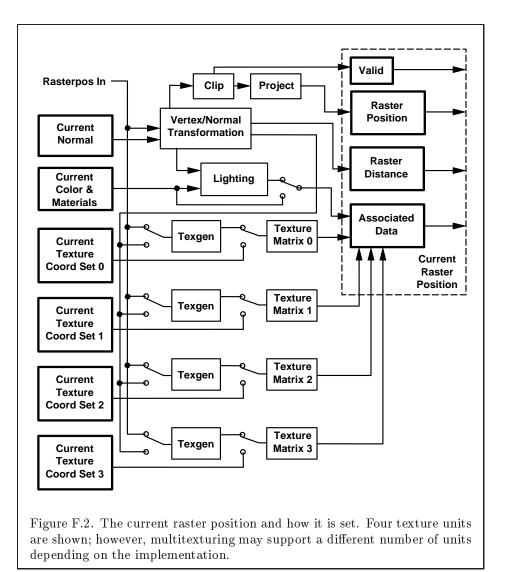
Amend paragraph 5

The current raster position requires five single-precision floating-point values for its x_w , y_w , and z_w window coordinates, its w_c clip coordinate, and its eye coordinate distance, a single valid bit, a color (RGBA and color index), and texture coordinates for each texture unit. In the initial state, the coordinates and texture coordinates are all (0, 0, 0, 1), the eye coordinate distance is 0, the valid bit is set, the associated RGBA color is (1, 1, 1, 1) and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color always maintains its initial value.

F.2.9 Changes to Section 3.8 (Texturing)

Amend paragraphs 1 and 2

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a fragment's (s, t, r)



coordinates to modify the fragment's primary RGBA color. Texturing does not affect the secondary color.

An implementation may support texturing using more than one image at a time. In this case the fragment carries multiple sets of texture coordinates (s, t, r) which are used to index separate images to produce color values which are collectively used to modify the fragment's RGBA color. Texturing is specified only for RGBA mode; its use in color index mode is undefined. The following subsections (up to and including Section 3.8.5) specify the GL operation with a single texture and Section 3.8.10 specifies the details of how multiple texture units interact.

F.2.10 Changes to Section 3.8.5 (Texture Minification)

Amend second paragraph under the Mipmapping subheading

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, CopyTexImage2D, TexImage1D, or CopyTexImage1D; the array being set is indicated with the level-of-detail argument level. Level-of-detail numbers proceed from TEXTURE_BASE_LEVEL for the original texture array through $p = \max\{n, m, l\}$ + TEXTURE_BASE_LEVEL with each unit increase indicating an array of half the dimensions of the previous one as already described. If texturing is enabled (and TEXTURE_MIN_FILTER is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays **TEXTURE_BASE_LEVEL** through $q = \min\{p, \text{TEXTURE_MAX_LEVEL}\}$ is incomplete. then it is as if texture mapping were disabled for that texture unit. The set of arrays **TEXTURE_BASE_LEVEL** through q is incomplete if the internal formats of all the mipmap arrays were not specified with the same symbolic constant, if the border widths of the mipmap arrays are not the same, if the dimensions of the mipmap arrays do not follow the sequence described above, if TEXTURE MAX_LEVEL < TEXTURE BASE_LEVEL, or if TEXTURE BASE_LEVEL > p. Array levels k where $k < \text{TEXTURE_BASE_LEVEL}$ or k > q are insignificant.

F.2.11 Changes to Section 3.8.8 (Texture Objects)

Insert following the last paragraph

The texture object name space, including the initial one-, two-, and three-dimensional texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state ACTIVE_TEXTURE_ARB.

If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

F.2.12 Changes to Section 3.8.10 (Texture Application)

Amend second paragraph

Each texture unit is enabled and bound to texture objects independently from the other texture units. Each texture unit follows the precendence rules for one-, two-, and three-dimensional textures. Thus texture units can be performing texture mapping of different dimensionalities simultaneously. Each unit has its own enable and binding states.

Each texture unit is paired with an environment function, as shown in figure F.3. The second texture function is computed using the texture value from the second texture, the fragment resulting from the first texture function computation and the second texture unit's environment function. If there is a third texture, the fragment resulting from the second texture function is combined with the third texture value using the third texture unit's environment function and so on. The texture unit selected by **ActiveTextureARB** determines which texture unit's environment is modified by **TexEnv** calls.

Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit, is passed unaltered to the following unit.

The required state, per texture unit, is three bits indicating whether each of one-, two-, or three-dimensional texturing is enabled or disabled. In the initial state, all texturing is disabled for all texture units.

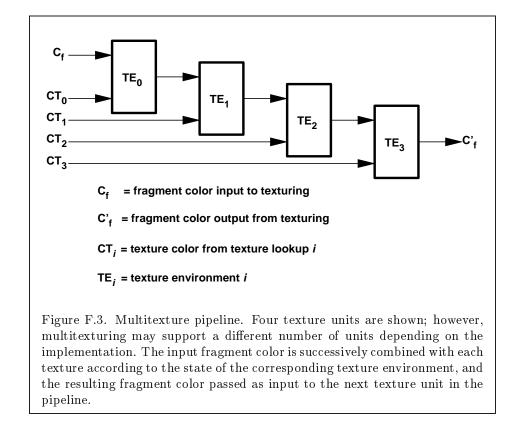
F.2.13 Changes to Section 5.1 (Evaluators)

Amend paragraph 7

The evaluation of a defined map is enabled or disabled with **Enable** and **Disable** using the constant corresponding to the map as described above. The evaluator map generates only coordinates for texture unit **TEXTUREO_ARB**. The error **INVALID_VALUE** results if either *ustride* or *vstride* is less than k, or if u_1 is equal to u_2 , or if v_1 is equal to v_2 . If the value of **ACTIVE_TEXTURE_ARB** is not **TEXTUREO_ARB**, calling **Map[12]** generates the error **INVALID_OPERATION**.

F.2.14 Changes to Section 5.3 (Feedback)

Amend paragraph 4



The texture coordinates and colors returned are those resulting from the clipping operations described in Section 2.13.8. Only coordinates for texture unit TEXTUREO_ARB are returned even for implementations which support multiple texture units. The colors returned are the primary colors.

F.2.15 Changes to Section 6.1.2 (Data Conversions)

Insert following the last paragraph

Most texture state variables are qualified by the value of ACTIVE_TEXTURE_ARB to determine which server texture state vector is queried. Client texture state variables such as texture coordinate array pointers are qualified by the value of CLIENT_ACTIVE_TEXTURE_ARB. Tables 6.5, 6.6, 6.7, 6.12, 6.14, and 6.25 indicate those state variables which are qualified by ACTIVE_TEXTURE_ARB or CLIENT_ACTIVE_TEXTURE_ARB during state queries.

F.2.16 Changes to Section 6.1.12 (Saving and Restoring State)

Insert following paragraph 3

Operations on groups containing replicated texture state push or pop texture state within that group for all texture units. When state for a group is pushed, all state corresponding to TEXTUREO_ARB is pushed first, followed by state corresponding to TEXTURE1_ARB, and so on up to and including the state corresponding to TEXTURE*_ARB where k + 1 is the value of MAX_TEXTURE_UNITS_ARB. When state for a group is popped, the replicated texture state is restored in the opposite order that it was pushed, starting with state corresponding to TEXTURE*_ARB and ending with TEXTUREO_ARB. Identical rules are observed for client texture state push and pop operations. Matrix stacks are never pushed or popped with PushAttrib, PushClientAttrib, PopAttrib, or PopClientAttrib.

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
Modified state in table 6.5						
CURRENT_TEXTURE_COORDS	1* imes T	GetFloatv	0,0,0,1	Current texture	2.7	current
				coordinates		
CURRENT_RASTER_TEXTURE_COORDS	1 * imes T	GetFloatv	0,0,0,1	Texture coordinates	2.12	current
				associated with		
				raster position		
Modified state in table 6.6						
TEXTURE-COORD-ARRAY	1* imes B	IsEnabled	False	Texture coordinate	2.8	vertex-array
				array enable		
TEXTURE_COORD_ARRAY_SIZE	$1* imes Z^+$	GetIntegerv	4	Coordinates per	2.8	vertex-array
				element		
TEXTURE_COORD_ARRAY_TYPE	$1* imes Z_4$	GetIntegerv	FLOAT	Type of texture	2.8	vertex-array
				coordinates		
TEXTURE-COORD_ARRAY_STRIDE	$1 * \times Z^+$	GetIntegerv	0	Stride between	2.8	vertex-array
				texture coordinates		
TEXTURE-COORD-ARRAY_POINTER	1* imes Y	GetPointerv	0	Pointer to the	2.8	vertex-array
				texture coordinate		
				array		

Table F.1. Changes to State Tables

Attribute	1	I		texture/enable	texture		texture	texture	texture/enable	texture	texture	texture
Sec.	2.10.2	2.10.2	-	3.8.10	3.8.8	-	3.8.9	3.8.9	2.10.4	2.10.4	2.10.4	2.10.4
Description	Texture matrix stack	Texture matrix stack pointer		True if xD texturing is enabled; x is 1, 2, or 3	Texture object bound to TEXTURE_xD		Texture application function	Texture environment color	Texgen enabled $(x \text{ is } S, T, R, \text{ or } Q)$	Texgen plane equation coefficients (for S, T, R, and Q)	Texgen object linear coefficients (for S, T, R, and Q)	Function used for texgen (for S, T, R, and Q
Initial Value	Identity	1		False	0		MODULATE	0,0,0,0	False	see 2.10.4	see 2.10.4	EYE_LINEAR
Get Cmnd	GetFloatv	GetIntegerv		IsEnabled	GetIntegerv		GetTexEnviv	GetTexEnvfv	IsEnabled	GetTexGenfv	GetTexGenfv	GetTexGeniv
$\begin{array}{c} {\rm Type} \\ {\rm e}~6.7 \end{array}$	$1* imes 2* imes M^4$	$1* imes Z^+$	e 6.12	1* imes 3 imes B	$1* imes 3 imes Z^+$	e 6.14	$1 * imes Z_4$	1 * imes C	$1*{ imes}4 imes B$	$1* imes 4 imes R^4$	$1* imes 4 imes R^4$	$1 * imes 4 imes Z_3$
Get value Modified state in table 6.7	TEXTURE_MATRIX	TEXTURE_STACK_DEPTH	Modified state in table 6.1	TEXTURE_ZD	TEXTURE_BINDING_&D	Modified state in table	TEXTURE_ENV_MODE	TEXTURE ENV COLOR	TEXTURE-GEN-x	EYE_PLANE	OBJECT_PLANE	TEXTURE_GEN_MODE

Table F.2. Changes to State Tables (cont.) Version 1.2.1 - April 1, 1999

			1			
Sec. Attribute	vertex-array			texture		
Sec.	2.7			2.7		
Description	Client active texture	unit selector		Z_{1*} GetIntegerv TEXTUREO_ARB Active texture unit	selector	
Initial Value	TEXTUREO_ARB			TEXTUREO_ARB		
Get Cmnd	GetIntegerv			GetIntegerv		
Type	Z_{1*}			Z_{1*}		
Get value Added to table 6.6	CLIENT-ACTIVE TEXTURE ARB Z_{1*} GetIntegerv TEXTUREO ARB Client active texture 2.7 vertex-array		Added to table 6.14	ACTIVE_TEXTURE_ARB		

Table F.3. New State Introduced by Multitexture

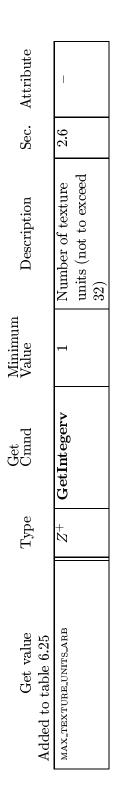


Table F.4. New Implementation-Dependent Values Introduced by Multitexture

Index of OpenGL Commands

*x*_BIAS, 78, 208 *x*_SCALE, 78, 208 2D, 174, 176, 217 2_BYTES, 177 3D, 174, 176 3D_COLOR, 174, 176 3D_COLOR_TEXTURE, 174, 176 3_BYTES, 177 4D_COLOR_TEXTURE, 174, 176 4_BYTES, 177

 $\begin{matrix} 1, \ 113, \ 120, \ 131, \ 136, \ 137, \ 185, \ 202, \\ 253 \end{matrix}$ $\begin{matrix} 2, \ 113, \ 120, \ 136, \ 137, \ 185, \ 202, \ 253 \\ 3, \ 113, \ 120, \ 136, \ 137, \ 185, \ 202, \ 253 \end{matrix}$

 $4,\,113,\,120,\,136,\,137,\,185$

ACCUM, 155 Accum, 155, 156 ACCUM_BUFFER_BIT, 154, 191 ACTIVE_TEXTURE_ARB, 243-246, 248, 249, 251 ActiveTextureARB, 244, 246, 249 ADD, 155, 156 ALL_ATTRIB_BITS, 191 ALL_CLIENT_ATTRIB_BITS, 191 ALPHA, 78, 92, 103, 104, 114, 115, 136, 137, 159, 160, 185, 208, 210, 216, 226, 232 ALPHA12, 115 ALPHA16, 115 ALPHA4, 115 ALPHA8, 115 ALPHA_BIAS, 101 ALPHA_SCALE, 101 ALPHA_TEST, 143

AlphaFunc, 143 ALWAYS, 143-145, 205 AMBIENT, 50, 51 AMBIENT_AND_DIFFUSE, 50, 51, 53AND, 151 AND_INVERTED, 151 AND_REVERSE, 151 AreTexturesResident, 134, 178 ArrayElement, 19, 23, 24, 175 AUTO_NORMAL, 167 AUXi, 151, 152 AUXn, 151, 158 AUX0, 151, 158 BACK, 49, 51, 52, 70, 73, 151, 152, 158, 159, 183, 201 BACK_LEFT, 151, 152, 158 BACK_RIGHT, 151, 152, 158 Begin, 12, 15-20, 23, 24, 28, 55, 62, 67, 70, 73, 168, 169, 174 BGR, 92, 159, 162 BGRA, 92, 94, 98, 159, 230 BindTexture, 133 BITMAP, 72, 80, 83, 90, 91, 98, 110, 160.185 Bitmap, 110 BITMAP_TOKEN, 176 BLEND, 135, 137, 146, 150 BlendColor, 77, 146 BlendEquation, 77, 146, 147 BlendFunc, 77, 146, 147, 149 BLUE, 78, 92, 159, 160, 208, 210, 216 BLUE_BIAS, 101 BLUE_SCALE, 101 BYTE, 22, 91, 160, 161, 177

C3F_V3F, 25, 26 C4F_N3F_V3F, 25, 26 C4UB_V2F, 25, 26 C4UB_V3F, 25, 26 CallList, 19, 177, 178 CallLists, 19, 177, 178 CCW, 48, 201 CLAMP, 124, 127 CLAMP_TO_EDGE, 124, 125, 127, 231**CLEAR**, 151 Clear, 153, 154 ClearAccum, 154 ClearColor. 154 ClearDepth, 154 ClearIndex, 154 ClearStencil, 154 CLIENT_ACTIVE_TEXTURE_ ARB, 243, 251 CLIENT_PIXEL_STORE_BIT, 191 CLIENT_VERTEX_ARRAY_BIT, 191ClientActiveTextureARB, 243 CLIP_PLANEi, 39 CLIP_PLANE0, 39 ClipPlane, 38 COEFF, 184 COLOR, 31, 34, 81, 85, 86, 120, 162, 245Color, 19-21, 43, 56 Color3, 20 Color4, 20 COLOR_ARRAY, 23, 27 COLOR_ARRAY_POINTER, 189 COLOR_BUFFER_BIT, 153, 191 COLOR_INDEX, 72, 80, 83, 90, 92, 102, 110, 159, 162, 184, 185 COLOR_INDEXES, 50, 54 COLOR_LOGIC_OP, 150 COLOR_MATERIAL, 51, 53 COLOR_MATRIX, 185 COLOR_MATRIX_STACK_DEPTH, 185COLOR_TABLE, 80, 82, 103 COLOR_TABLE_ALPHA_SIZE, 186

COLOR_TABLE_BIAS, 80, 81, 186 COLOR_TABLE_BLUE_SIZE, 186 COLOR_TABLE_FORMAT, 186 COLOR_TABLE_GREEN_SIZE, 186 COLOR_TABLE_INTENSITY_ SIZE, 186 COLOR_TABLE_LUMINANCE_ SIZE. 186 COLOR_TABLE_RED_SIZE, 186 COLOR_TABLE_SCALE, 80, 81, 186 COLOR_TABLE_WIDTH, 186 ColorMask, 152, 153 ColorMaterial, 51-53, 167, 223, 228 ColorPointer, 19, 21, 22, 27, 178 ColorSubTable, 81, 82 ColorTable, 79, 81-83, 108, 109, 179 ColorTableParameter, 80 ColorTableParameterfv, 80 Colorub, 56 Colorui, 56 Colorus, 56 COMPILE, 175, 223 COMPILE_AND_EXECUTE, 175,177, 178 CONSTANT_ALPHA, 77, 148, 149 CONSTANT_ATTENUATION, 50 CONSTANT_BORDER, 105, 106 CONSTANT_COLOR, 77, 148, 149 CONVOLUTION_1D, 84, 86, 103, 117, 186, 187 CONVOLUTION_2D, 83-85, 103, 117, 186, 187 CONVOLUTION_BORDER_ COLOR, 106, 187 CONVOLUTION_BORDER_ MODE, 105, 187 CONVOLUTION_FILTER_BIAS, 83-85, 187 CONVOLUTION_FILTER_SCALE, 83-86, 187 CONVOLUTION_FORMAT, 187 CONVOLUTION_HEIGHT, 187 CONVOLUTION_WIDTH, 187 ConvolutionFilter1D, 84-86 ConvolutionFilter2D, 83-86

ConvolutionParameter, 84, 105 ConvolutionParameterfv, 83, 84, 106 ConvolutionParameteriv, 85, 106 COPY, 150, 151, 205 COPY_INVERTED, 151 COPY_PIXEL_TOKEN, 176 CopyColorSubTable, 81, 82 CopyColorTable, 81, 82 CopyConvolutionFilter1D, 85 CopyConvolutionFilter2D, 85 CopyPixels, 75, 78, 81, 85, 86, 103, 120, 156, 162, 163, 173 CopyTexImage1D, 103, 120, 121, 129, 248CopyTexImage2D, 103, 118, 120, 121, 129, 248 CopyTexImage3D, 121 CopyTexSubImage1D, 103, 121, 123 CopyTexSubImage2D, 103, 121, 122 CopyTexSubImage3D, 103, 121, 122 CULL_FACE, 70 CullFace, 70 CURRENT_BIT, 191 CURRENT_RASTER_ TEXTURE_COORDS, 222, 246CURRENT_TEXTURE_COORDS, 243CW, 48 DECAL, 135, 137 **DECR**, 144 DeleteLists, 178 DeleteTextures, 133, 178 DEPTH, 162, 208 DEPTH_BIAS, 78, 101 DEPTH_BUFFER_BIT, 153, 191 DEPTH_COMPONENT, 80, 83, 90, 92, 112, 158, 159, 162, 184 DEPTH_SCALE, 78, 101 DEPTH_TEST, 145 DepthFunc, 145 DepthMask, 153 DepthRange, 30, 182, 224 **DIFFUSE**, 50, 51

Disable, 35, 38, 39, 44, 51, 60, 64, 67, 70, 72, 74, 108, 109, 138, 143-146, 149, 150, 166, 167, 246, 249 DisableClientState, 19, 23, 27, 178, 243DITHER, 150 DOMAIN, 184 DONT_CARE, 180, 213 DOUBLE, 22 DRAW_PIXEL_TOKEN, 176 DrawArrays, 23, 24, 175 DrawBuffer, 151, 152 DrawElements, 24, 25, 175, 232 DrawPixels, 72, 75, 76, 78, 80, 83, 89-93, 98, 100, 103, 110, 112, 113, 156, 158, 160, 162, 173 DrawRangeElements, 25, 215 DST_ALPHA, 148 DST_COLOR, 148 EDGE_FLAG_ARRAY, 23, 27 EDGE_FLAG_ARRAY_POINTER, 189EdgeFlag, 18, 19 EdgeFlagPointer, 19, 21, 22, 178 EdgeFlagv, 18 EMISSION, 50, 51 Enable, 35, 38, 39, 44, 51, 60, 64, 67, 70, 72, 74, 108, 109, 138, 143-146, 149, 150, 166, 167, 181, 246, 249 ENABLE_BIT, 191 EnableClientState, 19, 23, 27, 178, 243End, 12, 15-20, 23, 24, 28, 55, 62, 70, 73, 168, 169, 174 EndList, 175, 177 EQUAL, 143-145 **EQUIV**, 151 EVAL_BIT, 191 EvalCoord, 19, 167 EvalCoord1, 167-169 EvalCoord1d, 168 EvalCoord1f, 168

EvalCoord2, 167, 169, 170 EvalMesh1, 168 EvalMesh2, 168, 169 EvalPoint, 19 EvalPoint1, 169 EvalPoint2, 170 EXP, 139, 140, 198 EXP2, 139 EXT_bgra, 230 EXT_blend_color, 234 EXT_blend_logic_op, 226 EXT_blend_minmax, 234 EXT_blend_subtract, 234 EXT_color_subtable, 233 EXT_color_table, 233 EXT_convolution, 233 EXT_copy_texture, 227 EXT_draw_range_elements, 232 EXT_histogram, 234 EXT_packed_pixels, 231 EXT_polygon_offset, 226 EXT_rescale_normal, 231 EXT_separate_specular_color, 231 EXT_subtexture, 227 EXT_texture, 226, 227 EXT_texture3D, 230 EXT_texture_object, 227 EXT_vertex_array, 225 EXTENSIONS, 77, 189, 239 EYE_LINEAR, 37, 38, 183, 204, 245, 246, 253 EYE_PLANE, 37 FALSE, 18, 19, 46-48, 76, 78, 87, 88, 98, 101, 109, 110, 134, 158, 182, 184, 187, 188 FASTEST, 180 FEEDBACK, 171, 173, 174, 224 FEEDBACK_BUFFER_POINTER, 189FeedbackBuffer, 173, 174, 178 FILL, 73-75, 169, 201, 223, 226 Finish, 178, 179, 222 FLAT, 54, 223

FLOAT, 22, 26, 27, 91, 160, 161, 177, 196, 244, 252 Flush, 178, 179, 222 FOG, 138 Fog, 139, 140 FOG_BIT, 191 FOG_COLOR, 139 FOG_DENSITY, 139 FOG_END, 139 FOG_HINT, 180 FOG_INDEX, 140 FOG_MODE, 139, 140 FOG_START, 139 FRONT, 49, 51, 70, 73, 151, 152, 158, 159, 183FRONT_AND_BACK, 49, 51-53, 70, 73, 151, 152 FRONT_LEFT, 151, 152, 158 FRONT_RIGHT, 151, 152, 158 FrontFace, 48, 70 Frustum, 32, 33, 223 FUNC_ADD, 147, 149, 205 FUNC_REVERSE_SUBTRACT, 147 FUNC_SUBTRACT, 147 GenLists, 178 GenTextures, 133, 134, 178, 184 GEQUAL, 143–145 Get, 30, 178, 181, 182, 243, 246 GetBooleanv, 181, 182, 193 GetClipPlane, 182, 183 GetColorTable, 83, 158, 185 GetColorTableParameter, 186 GetConvolutionFilter, 158, 186 GetConvolutionParameter, 187 GetConvolutionParameteriv, 83, 84 GetDoublev, 181, 182, 193 GetError, 11 GetFloatv, 181, 182, 185, 193 GetHistogram, 88, 158, 187 GetHistogramParameter, 188 GetIntegerv, 25, 181, 182, 185, 193, 244

GetLight, 182, 183 GetMap, 183 GetMaterial, 182, 183 GetMinmax, 158, 188 GetMinmaxParameter, 188 GetPixelMap, 183 GetPointerv, 189 GetPolygonStipple, 185 GetSeparableFilter, 158, 186 GetString, 189 GetTexEnv, 182, 183 GetTexGen, 182, 183 GetTexImage, 103, 132, 184, 186–188 GetTexImage1D, 158 GetTexImage2D, 158 GetTexImage3D, 158 GetTexLevelParameter, 182, 183 GetTexParameter, 182, 183 GetTexParameterfv, 132, 134 GetTexParameteriv, 132, 134 $GL_ARB_multitexture, 240$ **GREATER**, 143–145 GREEN, 78, 92, 159, 160, 208, 210, 216GREEN_BIAS, 101 GREEN_SCALE, 101 Hint, 179 HINT_BIT, 191 HISTOGRAM, 87, 88, 109, 187, 188 Histogram, 87, 88, 109, 179 HISTOGRAM_ALPHA_SIZE, 188 HISTOGRAM_BLUE_SIZE, 188 HISTOGRAM_FORMAT, 188 HISTOGRAM_GREEN_SIZE, 188 HISTOGRAM_LUMINANCE_SIZE, 188HISTOGRAM_RED_SIZE, 188 HISTOGRAM_SINK, 188

HISTOGRAM_WIDTH, 188 HP_convolution_border_modes, 233 INCR, 144 INDEX, 216 Index, 19, 21 INDEX_ARRAY, 23, 27 INDEX_ARRAY_POINTER, 189 INDEX_LOGIC_OP, 150 INDEX_OFFSET, 78, 101, 208 INDEX_SHIFT, 78, 101, 208 IndexMask, 152, 153 IndexPointer, 19, 22, 178 InitNames, 171 INT, 22, 91, 160, 161, 177 INTENSITY, 87, 88, 103, 104, 114, 115, 136, 137, 185, 208, 226 INTENSITY12, 87, 88, 115 INTENSITY16, 87, 88, 115 INTENSITY4, 87, 88, 115 INTENSITY8, 87, 88, 115 Interleaved Arrays, 19, 25, 26, 178 INVALID_ENUM, 12, 13, 38, 49, 77, 83, 87, 88, 90, 120, 132, 184, 243, 244 INVALID_OPERATION, 13, 19, 77, 90, 94, 133, 151, 156, 158, 159, 171, 173, 175, 249 INVALID_VALUE, 12, 13, 22, 25, 30, 33, 49, 60, 64, 76, 78-80, 82-84, 87, 113, 114, 116, 121-123, 130, 134, 139, 143, 154, 165, 166, 168, 175, 177, 183, 184, 249INVERT, 144, 151 IsEnabled, 178, 181, 193 IsList, 178 IsTexture, 178, 184 KEEP, 144, 145, 205 LEFT, 151, 152, 158 LEQUAL, 143–145 LESS, 143-145, 205 Light, 49, 50 LIGHT*i*, 49, 51, 224 LIGHT0, 49 LIGHT_MODEL_AMBIENT, 50 LIGHT_MODEL_COLOR_ CONTROL, 50 LIGHT_MODEL_LOCAL_VIEWER,

50

LIGHT_MODEL_TWO_SIDE, 50

260

INDEX

LIGHTING, 44 LIGHTING_BIT, 191 LightModel, 49, 50 LINE, 73–75, 168, 169, 201, 226 LINE_BIT, 191 LINE_LOOP, 15 LINE_RESET_TOKEN, 176 LINE_SMOOTH, 64 LINE_SMOOTH_HINT, 180 LINE_STIPPLE, 67 LINE_STRIP, 15, 168 LINE_TOKEN, 176 LINEAR, 124, 127, 130, 131, 139 LINEAR_ATTENUATION, 50 LINEAR_MIPMAP_LINEAR, 124,129, 130 LINEAR_MIPMAP_NEAREST, 124, 129, 130 LINES, 16, 67 LineStipple, 66 LineWidth, 62 LIST_BIT, 191 ListBase, 178, 179 LOAD, 155 LoadIdentity, 31 LoadMatrix, 31, 32 LoadName, 171 LOGIC_OP, 150 LogicOp, 150, 151 LUMINANCE, 92, 99, 103, 104, 113-115, 136, 137, 159, 160, 185, 208, 210, 226 LUMINANCE12, 115 LUMINANCE12_ALPHA12, 115 LUMINANCE12_ALPHA4, 115 LUMINANCE16, 115 LUMINANCE16_ALPHA16, 115 LUMINANCE4, 115 LUMINANCE4_ALPHA4, 115 LUMINANCE6_ALPHA2, 115 LUMINANCE8, 115 LUMINANCE8_ALPHA8, 115 LUMINANCE_ALPHA, 92, 99, 103, 104, 113-115, 136, 137, 159, 160, 162, 185

Map1, 165, 166, 182 MAP1_COLOR_4, 165 MAP1_INDEX, 165 MAP1_NORMAL, 165 MAP1_TEXTURE_COORD_1, 165, 167MAP1_TEXTURE_COORD_2, 165, 167MAP1_TEXTURE_COORD_3, 165 MAP1_TEXTURE_COORD_4, 165 MAP1_VERTEX_3, 165 MAP1_VERTEX_4, 165 Map2, 165, 166, 182 MAP2_VERTEX_3, 167 MAP2_VERTEX_4, 167 Map[12], 249 MAP_COLOR, 78, 101, 102 MAP_STENCIL, 78, 102 MAP_VERTEX_3, 167 MAP_VERTEX_4, 167 MapGrid1, 168 MapGrid2, 168 Material, 19, 49, 50, 54, 223 MatrixMode, 31 MAX, 147 MAX_3D_TEXTURE_SIZE, 116 MAX_ATTRIB_STACK_DEPTH, 190MAX_CLIENT_ATTRIB_STACK_ DEPTH, 190 MAX_COLOR_MATRIX_STACK_ **DEPTH**, 185 MAX_CONVOLUTION_HEIGHT, 83, 187 MAX_CONVOLUTION_WIDTH, 83, 84, 187 MAX_ELEMENTS_INDICES, 25 MAX_ELEMENTS_VERTICES, 25 MAX_EVAL_ORDER, 165, 166 MAX_PIXEL_MAP_TABLE, 79, 101 MAX_TEXTURE_SIZE, 116 MAX_TEXTURE_UNITS_ARB, 240, 243, 244, 251 MIN, 147 MINMAX, 88, 109, 188

Minmax, 88, 110 MINMAX_FORMAT, 188 MINMAX_SINK, 188 MODELVIEW, 31, 34, 245 MODULATE, 135, 136 MULT, 155, 156 MultiTexCoord, 241 MultiTexCoordARB, 243, 244 MultMatrix, 31, 32 N3F_V3F, 25, 26 NAND, 151 NEAREST, 124, 127, 130, 131 NEAREST_MIPMAP_LINEAR, 124, 129 - 131NEAREST_MIPMAP_NEAREST, 124, 129–131 NEVER, 143-145 NewList, 175, 177, 178 NICEST, 180 NO_ERROR, 11, 12 NONE, 151, 152 NOOP, 151 NOR, 151 Normal, 19, 20 Normal3, 8, 9, 20 Normal3d, 8 Normal3dv, 9 Normal3f, 8 Normal3fv, 9 NORMAL_ARRAY, 23, 27 NORMAL_ARRAY_POINTER, 189 NORMALIZE, 35 NormalPointer, 19, 22, 27, 178 NOTEQUAL, 143-145 OBJECT_LINEAR, 37, 38, 183, 245, 246**OBJECT_PLANE**, 37 ONE, 148, 149, 205 ONE_MINUS_CONSTANT_ALPHA, 77, 148, 149 ONE_MINUS_CONSTANT_COLOR, 77, 148, 149 ONE_MINUS_DST_ALPHA, 148

ONE_MINUS_DST_COLOR, 148 ONE_MINUS_SRC_ALPHA, 148 ONE_MINUS_SRC_COLOR, 148 OR, 151 OR_INVERTED, 151 OR_REVERSE, 151 **ORDER**, 184 Ortho, 32, 33, 223 OUT_OF_MEMORY, 12, 13, 177 PACK_ALIGNMENT, 158, 207 PACK_IMAGE_HEIGHT, 158, 184, 207PACK_LSB_FIRST, 158, 207 PACK_ROW_LENGTH, 158, 207 PACK_SKIP_IMAGES, 158, 184, 207 PACK_SKIP_PIXELS, 158, 207 PACK_SKIP_ROWS, 158, 207 PACK_SWAP_BYTES, 158, 207 PASS_THROUGH_TOKEN, 176 PassThrough, 174 PERSPECTIVE_CORRECTION_ HINT, 180 PIXEL_MAP_A_TO_A, 79, 101 PIXEL_MAP_B_TO_B, 79, 101 PIXEL_MAP_G_TO_G, 79, 101 PIXEL_MAP_I_TO_A, 79, 102 PIXEL_MAP_I_TO_B, 79, 102 PIXEL_MAP_I_TO_G, 79, 102 PIXEL_MAP_I_TO_I, 79, 102 PIXEL_MAP_I_TO_R, 79, 102 PIXEL_MAP_R_TO_R, 79, 101 PIXEL_MAP_S_TO_S, 79, 102 PIXEL_MODE_BIT, 191 PixelMap, 75, 78, 79, 162 PixelStore, 19, 75, 76, 78, 158, 162, 178PixelTransfer, 75, 78, 107, 162 PixelZoom, 100 POINT, 73, 74, 168, 169, 201, 226 POINT_BIT, 191 POINT_SMOOTH, 60 POINT_SMOOTH_HINT, 180 POINT_TOKEN, 176 POINTS, 15, 168

PointSize, 60 POLYGON, 16, 19 POLYGON_BIT, 191 POLYGON_OFFSET_FILL, 74 POLYGON_OFFSET_LINE, 74 POLYGON_OFFSET_POINT, 74 POLYGON_SMOOTH, 70 POLYGON_SMOOTH_HINT, 180 POLYGON_STIPPLE, 72 POLYGON_STIPPLE_BIT, 191 POLYGON_TOKEN, 176 PolygonMode, 69, 73, 75, 171, 173 PolygonOffset, 74 PolygonStipple, 72 PopAttrib, 189, 190, 192, 224, 251 PopClientAttrib, 19, 178, 189, 190, 192, 251 PopMatrix, 34, 245 PopName, 171 **POSITION**, 50, 183 POST_COLOR_MATRIX_x_BIAS, 78POST_COLOR_MATRIX_x_SCALE, 78POST_COLOR_MATRIX_ALPHA_ BIAS, 108 POST_COLOR_MATRIX_ALPHA_ **SCALE**, 108 POST_COLOR_MATRIX_BLUE_ **BIAS**, 108 POST_COLOR_MATRIX_BLUE_ **SCALE**, 108 POST_COLOR_MATRIX_COLOR_ TABLE, 80, 109 POST_COLOR_MATRIX_GREEN_ **BIAS**, 108 POST_COLOR_MATRIX_GREEN_ **SCALE**, 108 POST_COLOR_MATRIX_RED_ **BIAS**, 108 POST_COLOR_MATRIX_RED_ **SCALE**, 108 POST_CONVOLUTION_x_BIAS, 78 POST_CONVOLUTION_x_SCALE,

POST_CONVOLUTION_ALPHA_ BIAS, 107 POST_CONVOLUTION_ALPHA_ **SCALE**, 107 POST_CONVOLUTION_BLUE_ BIAS, 107 POST_CONVOLUTION_BLUE_ **SCALE**, 107 POST_CONVOLUTION_COLOR_ TABLE, 80, 108 POST_CONVOLUTION_GREEN_ **BIAS**, 107 POST_CONVOLUTION_GREEN_ **SCALE**, 107 POST_CONVOLUTION_RED_ BIAS, 107 POST_CONVOLUTION_RED_ **SCALE**, 107 PrioritizeTextures, 134, 135 PROJECTION, 31, 34, 245 PROXY_COLOR_TABLE, 80, 82, 179PROXY_HISTOGRAM, 87, 88, 179, 188 PROXY_POST_COLOR_MATRIX_ COLOR_TABLE, 80, 179 PROXY_POST_CONVOLUTION_ COLOR_TABLE, 80, 179 PROXY_TEXTURE_1D, 117, 132, 179, 183 PROXY_TEXTURE_2D, 116, 132, 179, 183 PROXY_TEXTURE_3D, 112, 132, 179, 183 PushAttrib, 189, 190, 192, 251 PushClientAttrib, 19, 178, 189, 190, 192, 251 PushMatrix, 34, 245 PushName, 171 Q, 36, 38, 183 QUAD_STRIP, 17 QUADRATIC_ATTENUATION, 50 QUADS, 18, 19 R, 36, 38, 183

R3_G3_B2, 115 RasterPos, 41, 171, 223, 246 RasterPos2, 41, 246 RasterPos3, 41, 246 RasterPos4, 41, 246 ReadBuffer, 158, 159, 162 ReadPixels, 75, 78, 91-93, 103, 156-160, 162, 178, 184-186 Rect, 28, 70 RED, 78, 92, 159, 160, 208, 210, 216 RED_BIAS, 101 RED_SCALE, 101 REDUCE, 105, 107, 209 RENDER, 171, 172, 217 RENDERER, 189 RenderMode, 171-174, 178 REPEAT, 124, 125, 127, 128, 131, 203REPLACE, 135, 136, 144 REPLICATE_BORDER, 105, 106 RESCALE_NORMAL, 35 ResetHistogram, 187 ResetMinmax, 188 RETURN, 155, 156 RGB, 92, 94, 98, 103, 104, 113–115, 136, 137, 159, 162, 185, 226 RGB10, 115 RGB10_A2, 115 RGB12, 115 RGB16, 115 RGB4, 115 RGB5, 115 RGB5_A1, 115 RGB8, 115 RGBA, 81, 82, 85-88, 92, 94, 98, 103, 104, 113-115, 136, 137, 159, 162, 185, 208-211 RGBA12, 115 RGBA16, 115 RGBA2, 115 RGBA4, 115 RGBA8, 115 RIGHT, 151, 152, 158 Rotate, 32, 223

S, 36, 37, 183 Scale, 32, 33, 223 Scissor, 143 SCISSOR_BIT, 191 SCISSOR_TEST, 143 SELECT, 171, 172, 224 SelectBuffer, 171, 172, 178 SELECTION_BUFFER_POINTER, 189SEPARABLE_2D, 85, 103, 117, 187 SeparableFilter2D, 84 SEPARATE_SPECULAR_COLOR, 47SET, 151 SGI_color_matrix, 233 SGIS_multitexture, 238 SGIS_texture_edge_clamp, 231 SGIS_texture_lod, 232 ShadeModel. 54 SHININESS, 50 SHORT, 22, 91, 160, 161, 177 SINGLE_COLOR, 46, 47, 199 SMOOTH, 54, 198 **SPECULAR**, 50, 51 SPHERE_MAP, 37, 38 SPOT_CUTOFF, 50 SPOT_DIRECTION, 50, 183 SPOT_EXPONENT, 50 SRC_ALPHA, 148 SRC_ALPHA_SATURATE, 148 SRC_COLOR, 148 STACK_OVERFLOW, 13, 34, 171, 190, 245STACK_UNDERFLOW, 13, 34, 171, 190, 245 STENCIL, 162 STENCIL_BUFFER_BIT, 154, 191 STENCIL_INDEX, 80, 83, 90, 92, 100, 112, 156, 158, 159, 162, 184STENCIL_TEST, 144 StencilFunc, 144, 222 StencilMask, 153, 156, 223 StencilOp, 144, 145

264

T, 36, 183 T2F_C3F_V3F, 25, 26 T2F_C4F_N3F_V3F, 25, 26 T2F_C4UB_V3F, 25, 26 T2F_N3F_V3F, 25, 26 T2F_V3F, 25, 26 T4F_C4F_N3F_V4F, 25, 26 T4F_V4F, 25, 26 TABLE_TOO_LARGE, 13, 80, 87 TexCoord, 19, 20, 241, 243 TexCoord1, 20, 241 TexCoord2, 20, 241 TexCoord3, 20, 241 TexCoord4, 20, 241 TexCoordPointer, 19, 21, 22, 27, 178, 243TexEnv, 135, 249 TexGen, 36-38, 240, 246 TexImage, 121 TexImage1D, 76, 103, 105, 113, 117, 118, 120, 121, 129, 132, 179, 248TexImage2D, 76, 87, 88, 103, 105, 113, 116–118, 120, 121, 129, 132, 179, 248 TexImage3D, 76, 112–114, 116–118, 121, 129, 132, 178, 184, 248 TexParameter, 123 TexParameter[if], 126, 130 TexParameterf. 134 TexParameterfv, 134 TexParameteri, 134 TexParameteriv, 134 TexSubImage, 121 TexSubImage1D, 103, 121, 123 TexSubImage2D, 103, 120–122 TexSubImage3D, 120–122 TEXTURE, 31, 34, 244, 245 TEXTUREi_ARB, 241 TEXTURE0_ARB, 243, 245, 249, 251, 254 TEXTURE1_ARB, 251 TEXTURE_xD, 202, 253 TEXTURE_1D, 103, 117, 120, 121,

124, 132, 133, 138, 183, 184

TEXTURE_2D, 103, 116, 120, 121, 124, 132, 133, 138, 183, 184 TEXTURE_3D, 112, 121, 124, 132, 133, 138, 183, 184 TEXTURE_ALPHA_SIZE, 183 TEXTURE_BASE_LEVEL, 116, 117, 124, 126, 127, 129–132, 248 TEXTURE_BIT, 190, 191 TEXTURE_BLUE_SIZE, 183 TEXTURE_BORDER, 183 TEXTURE_BORDER_COLOR, 124, 129, 131, 132 TEXTURE_COMPONENTS, 183 TEXTURE_COORD_ARRAY, 23,27, 243 TEXTURE_COORD_ARRAY_ POINTER, 189 TEXTURE_DEPTH, 183 **TEXTURE_ENV**, 135, 183 TEXTURE_ENV_COLOR, 135 TEXTURE_ENV_MODE, 135 TEXTURE_GEN_MODE, 37, 38 TEXTURE_GEN_Q, 38 TEXTURE_GEN_R, 38 TEXTURE_GEN_S, 38 TEXTURE_GEN_T, 38 TEXTURE_GREEN_SIZE, 183 TEXTURE_HEIGHT, 183 TEXTURE_INTENSITY_SIZE, 183 TEXTURE_INTERNAL_FORMAT, 183TEXTURE_LUMINANCE_SIZE, 183TEXTURE_MAG_FILTER, 124, 131 TEXTURE_MAX_LEVEL, 116, 124, 130, 132, 248 TEXTURE_MAX_LOD, 124 - 126, 132TEXTURE_MIN_FILTER, 124, 127, 129–131, 248 TEXTURE_MIN_LOD, 124–126, 132 TEXTURE_PRIORITY, 124, 132, 134TEXTURE_RED_SIZE, 183 TEXTURE_RESIDENT, 132, 134

TEXTURE_WIDTH, 183 **TEXTURE_WRAP_R**, 124, 128 TEXTURE_WRAP_S, 124, 127, 128 **TEXTURE_WRAP_T**, 124, 128 TRANSFORM_BIT, 191 Translate, 32, 223 TRIANGLE_FAN, 17 TRIANGLE_STRIP, 16 TRIANGLES, 17, 19 TRUE, 18, 19, 40, 46-48, 76, 78, 87, 88, 134, 153, 158, 178, 182, 184, 187, 188 UNPACK_ALIGNMENT, 76, 93,112, 207 UNPACK_IMAGE_HEIGHT, 76,112, 207 UNPACK_LSB_FIRST, 76, 98, 207 UNPACK_ROW_LENGTH, 76, 90, 93, 112, 207 UNPACK_SKIP_IMAGES, 76, 112, 117, 207 UNPACK_SKIP_PIXELS, 76, 93, 98, 207UNPACK_SKIP_ROWS, 76, 93, 98, 207UNPACK_SWAP_BYTES, 76, 90, 92, 207UNSIGNED_BYTE, 22, 24, 26, 91, 95, 160, 161, 177 UNSIGNED_BYTE_2_3_3_REV, 91, 93-95, 161 UNSIGNED_BYTE_3_3_2, 91, 93-95, 161UNSIGNED_INT, 22, 24, 91, 97, 160, 161, 177 UNSIGNED_INT_10_10_10_2, 91, 94,

97, 161 UNSIGNED_INT_2_10_10_REV, 91, 94, 97, 161

- UNSIGNED_INT_8_8_8_8, 91, 94, 97, 161UNSIGNED_INT_8_8_8_REV, 91,
 - 94, 97, 161

UNSIGNED_SHORT, 22, 24, 91, 96, 160, 161, 177 UNSIGNED_SHORT_1_5_5_5_REV, 91, 94, 96, 161 UNSIGNED_SHORT_4_4_4_4, 91, 94, 96, 161 UNSIGNED_SHORT_4_4_4_REV, 91, 94, 96, 161 UNSIGNED_SHORT_5_5_5_1, 91, 94, 96, 161 UNSIGNED_SHORT_5_6_5, 91, 93, 94, 96, 161 UNSIGNED_SHORT_5_6_5_REV, 91, 93, 94, 96, 161 V2F, 25, 26 V3F, 25, 26 VENDOR, 189 VERSION, 189 Vertex, 7, 19, 20, 41, 167 Vertex2, 20, 28 Vertex2sv, 7 Vertex3, 20 Vertex3f, 7 Vertex4, 20 VERTEX_ARRAY, 23, 27 VERTEX_ARRAY_POINTER, 189 VertexPointer, 19, 22, 27, 178 Viewport, 30 VIEWPORT_BIT, 191 XOR, 151

ZERO, 144, 148, 149, 205

The OpenGL[®] Graphics System Utility Library (Version 1.3)

Norman Chin Chris Frazier Paul Ho Zicheng Liu Kevin P. Smith

Editor (version 1.3): Jon Leech

Copyright © 1992-1998 Silicon Graphics, Inc.

This document contains unpublished information of Silicon Graphics, Inc.

This document is protected by copyright, and contains information proprietary to Silicon Graphics, Inc. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of Silicon Graphics, Inc. is strictly prohibited. The receipt or possession of this document does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

U.S. Government Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227.7013 and/or in similar or successor clauses in the FAR or the DOD or NASA FAR Supplement. Unpublished rights reserved under the copyright laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

OpenGL is a registered trademark of Silicon Graphics, Inc. Unix is a registered trademark of The Open Group. The "X" device and X Windows System are trademarks of The Open Group.

Contents

1	Ove	rview	1				
2	Initi	alization	2				
3	Mip 3.1	mapping Image Scaling	4 4				
	3.2	Automatic Mipmapping	5				
4	Matrix Manipulation 7						
	4.1	Matrix Setup	7				
	4.2	Coordinate Projection	9				
5	Polygon Tessellation 10						
	5.1	The Tessellation Object	10				
	5.2	Polygon Definition	11				
	5.3	Callbacks	12				
	5.4	Control Over Tessellation	14				
	5.5	CSG Operations	16				
		5.5.1 UNION	17				
		5.5.2 INTERSECTION (two polygons at a time only)	17				
		5.5.3 DIFFERENCE	17				
	5.6	Performance	17				
	5.7	Backwards Compatibility	18				
6	Quadrics 20						
	6.1	The Quadrics Object	20				
	6.2		20				
	6.3	Rendering Styles	21				
	6.4		22				

CONTENTS

7	NURBS				
	7.1	The NURBS Object	24		
	7.2	Callbacks	25		
	7.3	NURBS Curves	27		
	7.4	NURBS Surfaces	27		
	7.5	Trimming	28		
	7.6	NURBS Properties	29		
8	\mathbf{Err}	ors	33		
9	\mathbf{GL}	U Versions	34		
	9.1	GLU 1.1	34		
	9.2	GLU 1.2	35		
	9.3	GLU 1.3	35		
	Ind	ex of GLU Commands	36		

ii

Chapter 1

Overview

The GL Utilities (GLU) library is a set of routines designed to complement the OpenGL graphics system by providing support for mipmapping, matrix manipulation, polygon tessellation, quadrics, NURBS, and error handling. Mipmapping routines include image scaling and automatic mipmap generation. A variety of matrix manipulation functions build projection and viewing matrices, or project vertices from one coordinate system to another. Polygon tessellation routines convert concave polygons into triangles for easy rendering. Quadrics support renders a few basic quadrics such as spheres and cones. NURBS code maps complicated NURBS curves and trimmed surfaces into simpler OpenGL evaluators. Lastly, an error lookup routine translates OpenGL and GLU error codes into strings. GLU library routines may call OpenGL library routines. Thus, an OpenGL context should be made current before calling any GLU functions. Otherwise an OpenGL error may occur.

All GLU routines, except for the initialization routines listed in Section 2, may be called during display list creation. This will cause any OpenGL commands that are issued as a result of the call to be stored in the display list. The result of calling the initialization routines after **glNewList** is undefined.

Chapter 2

Initialization

To get the GLU version number or supported GLU extensions call:

const GLubyte *gluGetString(GLenum name);

If *name* is GLU_VERSION or GLU_EXTENSIONS, then a pointer to a static zero-terminated string that describes the version or available extensions respectively is returned; otherwise NULL is returned.

The version string is laid out as follows:

<version number><space><vendor-specific information>

version number is either of the form major_number.minor_number or major_number.minor_number.release_number, where the numbers all have one or more digits. The version number determines which interfaces are provided by the GLU client library. If the underlying OpenGL implementation is an older version than that corresponding to this version of GLU, some of the GL calls made by GLU may fail. Chapter 9 describes how GLU versions and OpenGL versions correspond.

The vendor specific information is optional. However, if it is present the format and contents are implementation dependent.

The extension string is a space separated list of extensions to the GLU library. The extension names themselves do not contain any spaces. To determine if a specific extension name is present in the extension string, call

where extName is the extension name to check, and extString is the extension string. GL_TRUE is returned if extName is present in extString, GL_FALSE

otherwise. **gluCheckExtension** correctly handles boundary cases where one extension name is a substring of another. It may also be used to checking for the presence of OpenGL or GLX extensions by passing the extension strings returned by **glGetString** or **glXGetClientString**, instead of the GLU extension string.

gluGetString is not available in GLU 1.0. One way to determine whether this routine is present when using the X Window System is to query the GLX version. If the client version is 1.1 or greater then this routine is available. Operating system dependent methods may also be used to check for the existence of this function.

Chapter 3

Mipmapping

GLU provides image scaling and automatic mipmapping functions to simplify the creation of textures. The image scaling function can scale any image to a legal texture size. The resulting image can then be passed to OpenGL as a texture. The automatic mipmapping routines will take an input image, create mipmap textures from it, and pass them to OpenGL. With this interface, the user need only supply an image and the rest is automatic.

3.1 Image Scaling

The following routine magnifies or shrinks an image:

int gluScaleImage(GLenum format, GLsizei widthin, GLsizei heightin, GLenum typein, const void *datain, GLsizei widthout, GLsizei heightout, GLenum typeout, void *dataout);

gluScaleImage will scale an image using the appropriate pixel store modes to unpack data from the input image and pack the result into the output image. *format* specifies the image format used by both images. The input image is described by *widthin*, *heightin*, *typein*, and *datain*, where *widthin* and *heightin* specify the size of the image, *typein* specifies the data type used, and *datain* is a pointer to the image data in memory. The output image is similarly described by *widthout*, *heightout*, *typeout*, and *dataout*, where *widthout* and *heightout* specify the desired size of the image, *typeout* specifies the desired data type, and *dataout* points to the memory location where the image is to be stored. The pixel formats and types supported are the same as those supported by **glDrawPixels** for the underlying OpenGL implementation.

gluScaleImage reconstructs the input image by linear interpolation, convolves it with a one-pixel-square box kernel, and then samples the result to produce the output image.

A return value of 0 indicates success. Otherwise the return value is a GLU error code indicating the cause of the problem (see **gluErrorString** below).

3.2 Automatic Mipmapping

These routines will automatically generate mipmaps for any image provided by the user and then pass them to OpenGL:

```
int gluBuild1DMipmaps( GLenum target,
  GLint internalFormat, GLsizei width, GLenum format,
  GLenum type, const void *data );
```

```
int gluBuild2DMipmaps( GLenum target,
GLint internalFormat, GLsizei width, GLsizei height,
GLenum format, GLenum type, const void *data );
```

```
int gluBuild3DMipmaps( GLenum target,
    GLint internalFormat, GLsizei width, GLsizei height,
    GLsizei depth, GLenum format, GLenum type,
    const void *data );
```

gluBuild1DMipmaps, gluBuild2DMipmaps, and gluBuild3DMipmaps all take an input image and derive from it a pyramid of scaled images suitable for use as mipmapped textures. The resulting textures are then passed to glTexImage1D, glTexImage2D, or glTexImage3D as appropriate. *target, internalFormat, format, type, width, height, depth,* and *data* define the level 0 texture, and have the same meaning as the corresponding arguments to glTexImage1D, glTexImage2D, and glTexImage3D. Note that the image size does not need to be a power of 2, because the image will be automatically scaled to the nearest power of 2 size if necessary.

To load only a subset of mipmap levels, call

```
int gluBuild1DMipmapLevels( GLenum target,
GLint internalFormat, GLsizei width, GLenum format,
```

GLenum type, GLint level, GLint base, GLint max, const void *data);

int gluBuild2DMipmapLevels(GLenum target, GLint internalFormat, GLsizei width, GLsizei height, GLenum format, GLenum type, GLint level, GLint base, GLint max, const void *data);

int gluBuild3DMipmapLevels(GLenum target, GLint internalFormat, GLsizei width, GLsizei height,

GLsizei depth, GLenum format, GLenum type, GLint level, GLint base, GLint max, const void *data);

level specifies the mipmap level of the *input* image. *base* and *max* determine the minimum and maximum mipmap levels which will be passed to **glTexImagexD**. Other parameters are the same as for **gluBuildxDMipmaps**. If *level* > *base*, *base* < 0, *max* < *base*, or *max* is larger than the highest mipmap level for a texture of the specified size, no mipmap levels will be loaded, and the calls will return GLU_INVALID_VALUE.

A return value of 0 indicates success. Otherwise the return value is a GLU error code indicating the cause of the problem.

Chapter 4

Matrix Manipulation

The GLU library includes support for matrix creation and coordinate projection (transformation). The matrix routines create matrices and multiply the current OpenGL matrix by the result. They are used for setting projection and viewing parameters. The coordinate projection routines are used to transform object space coordinates into screen coordinates or vice-versa. This makes it possible to determine where in the window an object is being drawn.

4.1 Matrix Setup

The following routines create projection and viewing matrices and apply them to the current matrix using **glMultMatrix**. With these routines, a user can construct a clipping volume and set viewing parameters to render a scene.

 $\mathbf{gluOrtho2D}$ and $\mathbf{gluPerspective}$ build commonly-needed projection matrices.

void gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

sets up a two dimensional orthographic viewing region. The parameters define the bounding box of the region to be viewed. Calling **gluOrtho2D**(*left*, *right*, *bottom*, *top*) is equivalent to calling **glOrtho**(*left*, *right*, *bottom*, *top*, -1, 1).

```
void gluPerspective( GLdouble fory, GLdouble aspect,
GLdouble near, GLdouble far );
```

```
7
```

sets up a perspective viewing volume. fovy defines the field-of-view angle (in degrees) in the y direction. aspect is the aspect ratio used to determine the field-of-view in the x direction. It is the ratio of x (width) to y (height). *near* and *far* define the near and far clipping planes (as positive distances from the eye point).

gluLookAt creates a commonly-used viewing matrix:

void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

The viewing matrix created is based on an eye point (eyex, eyey, eyez), a reference point that represents the center of the scene (*centerx, centery, centerz*), and an up vector (upx, upy, upz). The matrix is designed to map the center of the scene to the negative Z axis, so that when a typical projection matrix is used, the center of the scene will map to the center of the viewport. Similarly, the projection of the up vector on the viewing plane is mapped to the positive Y axis so that it will point upward in the viewport. The up vector must not be parallel to the line-of-sight from the eye to the center of the scene.

gluPickMatrix is designed to simplify selection by creating a matrix that restricts drawing to a small region of the viewport. This is typically used to determine which objects are being drawn near the cursor. First restrict drawing to a small region around the cursor, then rerender the scene with selection mode turned on. All objects that were being drawn near the cursor will be selected and stored in the selection buffer.

void gluPickMatrix(GLdouble x, GLdouble y, GLdouble deltax, GLdouble deltay, const GLint viewport[4]);

gluPickMatrix should be called just before applying a projection matrix to the stack (effectively pre-multiplying the projection matrix by the selection matrix). x and y specify the center of the selection bounding box in pixel coordinates; *deltax* and *deltay* specify its width and height in pixels. *viewport* should specify the current viewport's x, y, width, and height. A convenient way to obtain this information is to call glGetIntegerv(GL_VIEWPORT, *viewport*).

4.2 Coordinate Projection

Two routines are provided to project coordinates back and forth from object space to screen space. **gluProject** projects from object space to screen space, and **gluUnProject** does the reverse. **gluUnProject4** should be used instead of **gluUnProject** when a nonstandard **glDepthRange** is in effect, or when a clip-space w coordinate other than 1 needs to be specified, as for vertices in the OpenGL **glFeedbackBuffer** when data type **GL_4D_COLOR_TEXTURE** is returned.

int gluProject(GLdouble objx, GLdouble objy, GLdouble objz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *winx, GLdouble *winy, GLdouble *winz);

gluProject performs the projection with the given *modelMatrix*, *projectionMatrix*, and *viewport*. The format of these arguments is the same as if they were obtained from glGetDoublev and glGetIntegerv. A return value of GL_TRUE indicates success, and GL_FALSE indicates failure.

int gluUnProject(GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *objx, GLdouble *objy, GLdouble *objz);

gluUnProject uses the given *modelMatrix*, *projectionMatrix*, and *view*port to perform the projection. A return value of GL_TRUE indicates success, and GL_FALSE indicates failure.

int gluUnProject4(GLdouble winx, GLdouble winy, GLdouble winz, GLdouble clipw, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLclampd near, GLclampd far, GLdouble *objx, GLdouble *objy, GLdouble *objz, GLdouble *objw);

gluUnProject4 takes three additional parameters and returns one additional parameter *clipw* is the clip-space w coordinate of the screen-space vertex (e.g. the w_c value computed by OpenGL); normally, clipw = 1. near and far correspond to the current glDepthRange; normally, near = 0 and far = 1. The object-space w value of the unprojected vertex is returned in objw. Other parameters are the same as for gluUnProject.

Chapter 5

Polygon Tessellation

The polygon tessellation routines triangulate concave polygons with one or more closed contours. Several winding rules are supported to determine which parts of the polygon are on the "interior". In addition, boundary extraction is supported: instead of tessellating the polygon, a set of closed contours separating the interior from the exterior are generated.

To use these routines, first create a tessellation object. Second, define the callback routines and the tessellation parameters. (The callback routines are used to process the triangles generated by the tessellator.) Finally, specify the concave polygon to be tessellated.

Input contours can be intersecting, self-intersecting, or degenerate. Also, polygons with multiple coincident vertices are supported.

5.1 The Tessellation Object

A new tessellation object is created with gluNewTess:

```
GLUtesselator *tessobj;
tessobj = gluNewTess(void);
```

gluNewTess returns a new tessellation object, which is used by the other tessellation functions. A return value of 0 indicates an out-of-memory error. Several tessellator objects can be used simultaneously.

When a tessellation object is no longer needed, it should be deleted with **gluDeleteTess**:

```
void gluDeleteTess( GLUtesselator *tessobj);
```

This will destroy the object and free any memory used by it.

5.2. POLYGON DEFINITION

5.2 Polygon Definition

The input contours are specified with the following routines:

```
void gluTessBeginPolygon(GLUtesselator *tess,
    void *polygon_data);
void gluTessBeginContour(GLUtesselator *tess);
void gluTessVertex(GLUtesselator *tess,
    GLdouble coords[3], void *vertex_data);
void gluTessEndContour(GLUtesselator *tess);
void gluTessEndPolygon(GLUtesselator *tess);
```

Within each gluTessBeginPolygon / gluTessEndPolygon pair, there must be one or more calls to gluTessBeginContour / gluTessEnd-Contour. Within each contour, there are zero or more calls to gluTessVertex. The vertices specify a closed contour (the last vertex of each contour is automatically linked to the first).

polygon_data is a pointer to a user-defined data structure. If the appropriate callback(s) are specified (see section 5.3), then this pointer is returned to the callback function(s). Thus, it is a convenient way to store per-polygon information.

coords give the coordinates of the vertex in 3-space. For useful results, all vertices should lie in some plane, since the vertices are projected onto a plane before tessellation. *vertex_data* is a pointer to a user-defined vertex structure, which typically contains other vertex information such as color, texture coordinates, normal, etc. It is used to refer to the vertex during rendering.

When **gluTessEndPolygon** is called, the tessellation algorithm determines which regions are interior to the given contours, according to one of several "winding rules" described below. The interior regions are then tessellated, and the output is provided as callbacks.

gluTessBeginPolygon indicates the start of a polygon, and it must be called first. It is an error to call gluTessBeginContour outside of a gluTessBeginPolygon / gluTessEndPolygon pair; it is also an error to call gluTessVertex outside of a gluTessBeginContour / gluTessEnd-Contour pair. In addition, gluTessBeginPolygon / gluTessEndPolygon and gluTessBeginContour / gluTessEndContour calls must pair up.

5.3 Callbacks

Callbacks are specified with gluTessCallback:

```
void gluTessCallback( GLUtesselator *tessobj,
GLenum which, void (*fn );())
```

This routine replaces the callback selected by which with the function specified by fn. If fn is equal to NULL, then any previously defined callback is discarded and becomes undefined. Any of the callbacks may be left undefined; if so, the corresponding information will not be supplied during rendering. (Note that, under some conditions, it is an error to leave the combine callback undefined. See the description of this callback below for details.)

It is legal to leave any of the callbacks undefined. However, the information that they would have provided is lost.

which mav be one of GLU_TESS_BEGIN. GLU_TESS_EDGE_FLAG. GLU_TESS_VERTEX, GLU_TESS_END, GLU_TESS_ERROR, GLU_TESS_COMBINE, GLU_TESS_BEGIN_DATA, GLU_TESS_EDGE_FLAG_DATA, GLU_TESS_VERTEX_DATA, GLU_TESS_END_DATA, GLU_TESS_ERROR_DATA or GLU_TESS_COMBINE_DATA. The twelve callbacks have the following prototypes:

```
void begin( GLenum type );
void edgeFlag( GLboolean flag );
void vertex( void *vertex_data );
void end( void );
void error( GLenum errno );
void combine( GLdouble coords[3], void *vertex_data[4],
GLfloat weight[4], void **outData );
void beginData( GLenum type, void *polygon_data );
void edgeFlagData( GLboolean flag, void *polygon_data );
void endData( void *polygon_data );
void vertexData( void *vertex_data, void *polygon_data );
void vertexData( GLenum errno, void *polygon_data );
void errorData( GLenum errno, void *polygon_data );
void combineData( GLdouble coords[3],
void *vertex_data[4], GLfloat weight[4], void **outDatab,
void *polygon_data );
```

Note that there are two versions of each callback: one with user-specified polygon data and one without. If both versions of a particular callback are

12

5.3. CALLBACKS

specified then the callback with *polygon_data* will be used. Note that *poly-gon_data* is a copy of the pointer that was specified when **gluTessBegin-Polygon** was called.

The begin callbacks indicate the start of a primitive. *type* is one of GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, or GL_TRIANGLES (but see the description of the edge flag callbacks below and the notes on *boundary extraction* in section 5.4 where the GLU_TESS_BOUNDARY_ONLY property is described).

It is followed by any number of **vertex** callbacks, which supply the vertices in the same order as expected by the corresponding **glBegin** call. *vertex_data* is a copy of the pointer that the user provided when the vertex was specified (see **gluTessVertex**). After the last vertex of a given primitive, the **end** or **endData** callback is called.

If one of the edge flag callbacks is provided, no triangle fans or strips will be used. When edgeFlag or edgeFlagData is called, if *flag* is GL_TRUE, then each vertex which follows begins an edge which lies on the polygon boundary (i.e., an edge which separates an interior region from an exterior one). If *flag* is GL_FALSE, each vertex which follows begins an edge which lies in the polygon interior. The edge flag callback will be called before the first call to the vertex callback.

The error or errorData callback is invoked when an error is encountered. The errno will be set to one of GLU_TESS_MISSING_BEGIN_POLYGON, GLU_TESS_MISSING_END_POLYGON, GLU_TESS_MISSING_BEGIN_CONTOUR, GLU_TESS_MISSING_END_CONTOUR, GLU_TESS_COORD_TOO_LARGE, or GLU_TESS_NEED_COMBINE_CALLBACK.

The first four errors are self-explanatory. The GLU library will recover from these errors by inserting the missing call(s). GLU_TESS_COORD_TOO_LARGE says that some vertex coordinate exceeded the predefined constant GLU_TESS_MAX_COORD_TOO_LARGE in absolute value, and that the value has been clamped. (Coordinate values must be small enough so that two can be multiplied together without overflow.) GLU_TESS_NEED_COMBINE_CALLBACK says that the algorithm detected an intersection between two edges in the input data, and the *combine* callback (below) was not provided. No output will be generated.

The combine or combineData callback is invoked to create a new vertex when the algorithm detects an intersection, or wishes to merge features. The vertex is defined as a linear combination of up to 4 existing vertices, referenced by $vertex_data[0..3]$. The coefficients of the linear combination are given by weight[0..3]; these weights always sum to 1.0. All vertex pointers are valid even when some of the weights are zero. *coords* gives the location of the new vertex. The user must allocate another vertex, interpolate parameters using vertex_data and weights, and return the new vertex pointer in outData. This handle is supplied during rendering callbacks. For example, if the polygon lies in an arbitrary plane in 3-space, and we associate a color with each vertex, the combine callback might look like this:

```
void MyCombine(GLdouble coords[3], VERTEX *d[4],
     GLfloat w[4], VERTEX **dataOut);
{
     VERTEX *new = new_vertex();
     new - x = coords[0];
     new ->y = coords[1];
     new ->z = coords[2];
     new ->r = w[0]*d[0] ->r + w[1]*d[1] ->r +
              w[2]*d[2] ->r + w[3]*d[3] ->r;
     new - g = w[0] * d[0] - g + w[1] * d[1] - g +
              w[2]*d[2]->g + w[3]*d[3]->g;
     new - b = w[0] * d[0] - b + w[1] * d[1] - b +
              w[2]*d[2]->b + w[3]*d[3]->b;
     new->a = w[0]*d[0]->a + w[1]*d[1]->a +
              w[2]*d[2]->a + w[3]*d[3]->a;
     *dataOut = new;
}
```

If the algorithm detects an intersection, then the combine or combineData callback must be defined, and it must write a non-NULL pointer into *dataOut*. Otherwise the GLU_TESS_NEED_COMBINE_CALLBACK error occurs, and no output is generated. This is the only error that can occur during tessellation and rendering.

5.4 Control Over Tessellation

The properties associated with a tessellator object affect the way the polygons are interpreted and rendered. The properties are set by calling:

void gluTessProperty(GLUtesselator tess, GLenum which, GLdouble value); which indicates the property to be modified and must be set to one of GLU_TESS_WINDING_RULE, GLU_TESS_BOUNDARY_ONLY, or GLU_TESS_TOLERANCE.

value specifies the new property

The GLU_TESS_WINDING_RULE property determines which parts of the polygon are on the *interior*. It is an enumerated value; the possible values are: GLU_TESS_WINDING_ODD, GLU_TESS_WINDING_NONZERO, GLU_TESS_WINDING_NEGATIVE, GLU_TESS_WINDING_POSITIVE and GLU_TESS_WINDING_ABS_GEQ_TWO.

To understand how the winding rule works first consider that the input contours partition the plane into regions. The winding rule determines which of these regions are inside the polygon.

For a single contour C, the winding number of a point \mathbf{x} is simply the signed number of revolutions we make around \mathbf{x} as we travel once around C, where counter-clockwise (CCW) is positive. When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point \mathbf{x} in the plane. Note that the winding number is the same for all points in a single region.

The winding rule classifies a region as *inside* if its winding number belongs to the chosen category (odd, nonzero, positive, negative, or absolute value of at least two). The previous GLU tessellator (prior to GLU 1.2) used the *odd* rule. The *nonzero* rule is another common way to define the interior. The other three rules are useful for polygon CSG operations (see below).

The GLU_TESS_BOUNDARY_ONLY property is a boolean value (value should be set to GL_TRUE or GL_FALSE). When set to GL_TRUE, a set of closed contours separating the polygon interior and exterior are returned instead of a tessellation. Exterior contours are oriented CCW with respect to the normal, interior contours are oriented clockwise (CW). The GLU_TESS_BEGIN and GLU_TESS_BEGIN_DATA callbacks use the type GL_LINE_LOOP for each contour.

GLU_TESS_TOLERANCE specifies a tolerance for merging features to reduce the size of the output. For example, two vertices which are very close to each other might be replaced by a single vertex. The tolerance is multiplied by the largest coordinate magnitude of any input vertex; this specifies the maximum distance that any feature can move as the result of a single merge operation. If a single feature takes part in several merge operations, the total distance moved could be larger.

Feature merging is completely optional; the tolerance is only a hint. The implementation is free to merge in some cases and not in others, or to never merge features at all. The default tolerance is zero.

The current implementation merges vertices only if they are exactly coincident, regardless of the current tolerance. A vertex is spliced into an edge only if the implementation is unable to distinguish which side of the edge the vertex lies on. Two edges are merged only when both endpoints are identical.

Property values can also be queried by calling

void gluGetTessProperty(GLUtesselator tess, GLenum which, GLdouble *value);

to load *value* with the value of the property specified by *which*. To supply the polygon normal call:

```
void gluTessNormal( GLUtesselator tess, GLdouble x,
GLdouble y, GLdouble z);
```

All input data will be projected into a plane perpendicular to the normal before tessellation and all output triangles will be oriented CCW with respect to the normal (CW orientation can be obtained by reversing the sign of the supplied normal). For example, if you know that all polygons lie in the x-y plane, call **gluTessNormal**(tess,0.0,0.0,1.0) before rendering any polygons.

If the supplied normal is (0,0,0) (the default value), the normal is determined as follows. The direction of the normal, up to its sign, is found by fitting a plane to the vertices, without regard to how the vertices are connected. It is expected that the input data lies approximately in plane; otherwise projection perpendicular to the computed normal may substantially change the geometry. The sign of the normal is chosen so that the sum of the signed areas of all input contours is non-negative (where a CCW contour has positive area).

The supplied normal persists until it is changed by another call to **gluTessNormal**.

5.5 CSG Operations

The features of the tessellator make it easy to find the union, difference, or intersection of several polygons.

First, assume that each polygon is defined so that the winding number is 0 for each exterior region, and 1 for each interior region. Under this model, CCW contours define the outer boundary of the polygon, and CW contours

define holes. Contours may be nested, but a nested contour must be oriented oppositely from the contour that contains it.

If the original polygons do not satisfy this description, they can be converted to this form by first running the tessellator with the GLU_TESS_BOUNDARY_ONLY property turned on. This returns a list of contours satisfying the restriction above. By allocating two tessellator objects, the callbacks from one tessellator can be fed directly to the input of another.

Given two or more polygons of the form above, CSG operations can be implemented as follows:

5.5.1 UNION

Draw all the input contours as a single polygon. The winding number of each resulting region is the number of original polygons which cover it. The union can be extracted using the GLU_TESS_WINDING_NONZERO or GLU_TESS_WINDING_POSITIVE winding rules. Note that with the nonzero rule, we would get the same result if all contour orientations were reversed.

5.5.2 INTERSECTION (two polygons at a time only)

Draw a single polygon using the contours from both input polygons. Extract the result using GLU_TESS_WINDING_ABS_GEQ_TWO. (Since this winding rule looks at the absolute value, reversing all contour orientations does not change the result.)

5.5.3 DIFFERENCE

Suppose we want to compute $A - (B \cup C \cup D)$. Draw a single polygon consisting of the unmodified contours from A, followed by the contours of B, C, and D with the vertex order reversed (this changes the winding number of the interior regions to -1). To extract the result, use the GLU_TESS_WINDING_POSITIVE rule.

If B, C, and D are the result of a GLU_TESS_BOUNDARY_ONLY call, an alternative to reversing the vertex order is to reverse the sign of the supplied normal. For example in the x-y plane, call gluTessNormal(tess, 0, 0, -1).

5.6 Performance

The tessellator is not intended for immediate-mode rendering; when possible the output should be cached in a user structure or display list. General polygon tessellation is an inherently difficult problem, especially given the goal of extreme robustness.

Single-contour input polygons are first tested to see whether they can be rendered as a triangle fan with respect to the first vertex (to avoid running the full decomposition algorithm on convex polygons). Non-convex polygons may be rendered by this "fast path" as well, if the algorithm gets lucky in its choice of a starting vertex.

For best performance follow these guidelines:

- supply the polygon normal, if available, using **gluTessNormal**. For example, if all polygons lie in the x-y plane, use **gluTessNormal**(*tess*, 0, 0, 1).
- render many polygons using the same tessellator object, rather than allocating a new tessellator for each one. (In a multi-threaded, multiprocessor environment you may get better performance using several tessellators.)

5.7 Backwards Compatibility

The polygon tessellation routines described previously are new in version 1.2 of the GLU library. For backwards compatibility, earlier versions of these routines are still supported:

```
void gluBeginPolygon( GLUtesselator *tess );
void gluNextContour( GLUtesselator *tess,
GLenum type );
```

```
void gluEndPolygon( GLUtesselator *tess );
```

gluBeginPolygon indicates the start of the polygon and gluEndPolygon defines the end of the polygon. gluNextContour is called once before each contour; however it does not need to be called when specifying a polygon with one contour. *type* is ignored by the GLU tessellator. *type* is one of GLU_EXTERIOR, GLU_INTERIOR, GLU_CCW, GLU_CW or GLU_UNKNOWN.

Calls to **gluBeginPolygon**, **gluNextContour** and **gluEndPolygon** are mapped to the new tessellator interface as follows:

${f gluBeginPolygon} ightarrow$	${f gluTessBeginPolygon}$
	${f gluTessBeginContour}$
$\mathbf{gluNextContour} \rightarrow$	${f gluTessEndContour}$
	${f gluTessBeginContour}$
${\bf gluEndPolygon} \rightarrow$	${f gluTessEndContour}$
	gluTessEndPolygon

GLU_ESS_ENDPOLYGON Constants and data structures used in the previous versions of the tessellator are also still supported. GLU_BEGIN, GLU_VERTEX, GLU_END, GLU_ERROR and GLU_EDGE_FLAG are defined as synonyms for GLU_TESS_BEGIN, GLU_TESS_VERTEX, GLU_TESS_END, GLU_TESS_ERROR and GLU_TESS_EDGE_FLAG. GLUTriangulatorObj is defined to be the same as GLUtesselator.

The preferred interface for polygon tessellation is the one described in sections 5.1-5.4. The routines described in this section are provided for backward compatibility only.

Chapter 6

Quadrics

The GLU library quadrics routines will render spheres, cylinders and disks in a variety of styles as specified by the user. To use these routines, first create a quadrics object. This object contains state indicating how a quadric should be rendered. Second, modify this state using the function calls described below. Finally, render the desired quadric by invoking the appropriate quadric rendering routine.

6.1 The Quadrics Object

A quadrics object is created with gluNewQuadric:

GLUquadricObj *quadobj; quadobj = gluNewQuadric(void);

gluNewQuadric returns a new quadrics object. This object contains state describing how a quadric should be constructed and rendered. A return value of 0 indicates an out-of-memory error.

When the object is no longer needed, it should be deleted with **gluDeleteQuadric**:

```
void gluDeleteQuadric( GLUquadricObj *quadobj );
```

This will delete the quadrics object and any memory used by it.

6.2 Callbacks

To associate a callback with the quadrics object, use **gluQuadricCallback**:

6.3. RENDERING STYLES

void gluQuadricCallback(GLUquadricObj *quadobj, GLenum which, void (*fn);())

The only callback provided for quadrics is the GLU_ERROR callback (identical to the polygon tessellation callback described above). This callback takes an error code as its only argument. To translate the error code to an error message, see gluErrorString below.

6.3 Rendering Styles

A variety of variables control how a quadric will be drawn. These are *nor-mals*, *texture Coords*, *orientation*, and *drawStyle*. *normals* indicates if surface normals should be generated, and if there should be one normal per vertex or one normal per face. *texture Coords* determines whether texture coordinates should be generated. *orientation* describes which side of the quadric should be the "outside". Lastly, *drawStyle* indicates if the quadric should be drawn as a set of polygons, lines, or points.

To specify the kind of normals desired, use gluQuadricNormals:

```
void gluQuadricNormals( GLUquadricObj *quadobj,
GLenum normals );
```

normals is either GLU_NONE (no normals), GLU_FLAT (one normal per face) or GLU_SMOOTH (one normal per vertex). The default is GLU_SMOOTH.

Texture coordinate generation can be turned on and off with gluQuadricTexture:

```
void gluQuadricTexture( GLUquadricObj *quadobj,
GLboolean textureCoords );
```

If *textureCoords* is GL_TRUE, then texture coordinates will be generated when a quadric is rendered. Note that how texture coordinates are generated depends upon the specific quadric. The default is GL_FALSE.

An orientation can be specified with gluQuadricOrientation:

```
void gluQuadricOrientation(GLUquadricObj *quadobj,
GLenum orientation);
```

If orientation is GLU_OUTSIDE then quadrics will be drawn with normals pointing outward. If orientation is GLU_INSIDE then the normals will point inward (faces are rendered counter-clockwise with respect to the normals).

Note that "outward" and "inward" are defined by the specific quadric. The default is GLU_OUTSIDE.

A drawing style can be chosen with **gluQuadricDrawStyle**:

```
void gluQuadricDrawStyle( GLUquadricObj *quadobj,
GLenum drawStyle );
```

drawStyle is one of GLU_FILL, GLU_LINE, GLU_POINT or GLU_SILHOUETTE. In GLU_FILL mode, the quadric is rendered as a set of polygons, in GLU_LINE mode as a set of lines, and in GLU_POINT mode as a set of points. GLU_SILHOUETTE mode is similar to GLU_LINE mode except that edges separating coplanar faces are not drawn. The default style is GLU_FILL.

6.4 Quadrics Primitives

The four supported quadrics are spheres, cylinders, disks, and partial disks. Each of these quadrics may be subdivided into arbitrarily small pieces.

A sphere can be created with **gluSphere**:

```
void gluSphere( GLUquadricObj *quadobj,
GLdouble radius, GLint slices, GLint stacks );
```

This renders a sphere of the given *radius* centered around the origin. The sphere is subdivided along the Z axis into the specified number of *stacks*, and each stack is then sliced evenly into the given number of *slices*. Note that the globe is subdivided in an analogous fashion, where lines of latitude represent *stacks*, and lines of longitude represent *slices*.

If texture coordinate generation is enabled then coordinates are computed so that t ranges from 0.0 at Z = -radius to 1.0 at Z = radius (t increases linearly along longitudinal lines), and s ranges from 0.0 at the +Y axis, to 0.25 at the +X axis, to 0.5 at the -Y axis, to 0.75 at the -X axis, and back to 1.0 at the +Y axis.

A cylinder is specified with **gluCylinder**:

void gluCylinder(GLUquadricObj *quadobj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, GLint slices, GLint stacks);

gluCylinder draws a frustum of a cone centered on the Z axis with the base at Z = 0 and the top at Z = height. base Radius specifies the radius at Z

22

= 0, and topRadius specifies the radius at Z = height. (If baseRadius equals topRadius, the result is a conventional cylinder.) Like a sphere, a cylinder is subdivided along the Z axis into stacks, and each stack is further subdivided into slices. When textured, t ranges linearly from 0.0 to 1.0 along the Z axis, and s ranges from 0.0 to 1.0 around the Z axis (in the same manner as it does for a sphere).

A disk is created with **gluDisk**:

```
void gluDisk( GLUquadricObj *quadobj,
GLdouble innerRadius, GLdouble outerRadius,
GLint slices, GLint loops );
```

This renders a disk on the Z=0 plane. The disk has the given *outer-Radius*, and if *innerRadius* > 0.0 then it will contain a central hole with the given *innerRadius*. The disk is subdivided into the specified number of *slices* (similar to cylinders and spheres), and also into the specified number of *loops* (concentric rings about the origin). With respect to orientation, the +Z side of the disk is considered to be "outside".

When textured, coordinates are generated in a linear grid such that the value of (s,t) at (outerRadius,0,0) is (1,0.5), at (0,outerRadius,0) it is (0.5,1), at (-outerRadius,0,0) it is (0,0.5), and at (0,-outerRadius,0) it is (0.5,0). This allows a 2D texture to be mapped onto the disk without distortion.

A partial disk is specified with **gluPartialDisk**:

void gluPartialDisk(GLUquadricObj *quadobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops, GLdouble startAngle, GLdouble sweepAngle);

This function is identical to gluDisk except that only the subset of the disk from *startAngle* through *startAngle* + *sweepAngle* is included (where 0 degrees is along the +Y axis, 90 degrees is along the +X axis, 180 is along the -Y axis, and 270 is along the -X axis). In the case that *drawStyle* is set to either GLU_FILL or GLU_SILHOUETTE, the edges of the partial disk separating the included area from the excluded arc will be drawn.

Chapter 7

NURBS

NURBS curves and surfaces are converted to OpenGL primitives by the functions in this section. The interface employs a NURBS object to describe the curves and surfaces and to specify how they should be rendered. Basic trimming support is included to allow more flexible definition of surfaces.

There are two ways to handle a NURBS object (curve or surface), to either render or to tessellate. In rendering mode, the objects are converted or tessellated to a sequence of OpenGL evaluators and sent to the OpenGL pipeline for rendering. In tessellation mode, objects are converted to a sequence of triangles and triangle strips and returned back to the application through a callback interface for further processing. The decomposition algorithm used for rendering and for returning tessellations are not guaranteed to produce identical results.

7.1 The NURBS Object

A NURBS object is created with gluNewNurbsRenderer:

```
GLUnurbsObj *nurbsObj;
nurbsObj = gluNewNurbsRenderer(void);
```

nurbsObj is an opaque pointer to all of the state information needed to tessellate and render a NURBS curve or surface. Before any of the other routines in this section can be used, a NURBS object must be created. A return value of 0 indicates an out of memory error.

When a NURBS object is no longer needed, it should be deleted with **gluDeleteNurbsRenderer**:

```
void gluDeleteNurbsRenderer( GLUnurbsObj *nurbsObj );
```

This will destroy all state contained in the object, and free any memory used by it.

7.2 Callbacks

To define a callback for a NURBS object, use:

```
void gluNurbsCallback( GLUnurbsObj *nurbsObj,
GLenum which, void (*fn );())
```

The parameter *which* can be one of the following: GLU_NURBS_BEGIN, GLU_NURBS_VERTEX, GLU_NORMAL, GLU_NURBS_COLOR, GLU_NURBS_TEXTURE_COORD, GLU_END, GLU_NURBS_BEGIN_DATA, GLU_NURBS_VERTEX_DATA, GLU_NORMAL_DATA, GLU_NURBS_COLOR_DATA, GLU_NURBS_TEXTURE_COORD_DATA, GLU_END_DATA and GLU_ERROR.

These callbacks have the following prototypes:

```
void begin( GLenum type );
void vertex( GLfloat *vertex );
void normal( GLfloat *normal );
void color( GLfloat *color );
void texCoord( GLfloat *tex_coord );
void end( void );
void beginData( GLenum type, void *userData );
void vertexData( GLfloat *vertex, void *userData );
void normalData( GLfloat *normal, void *userData );
void colorData( GLfloat *color, void *userData );
void texCoordData( GLfloat *tex_coord, void *userData );
void endData( void *userData );
void endData( void *userData );
```

The first 12 callbacks are for the user to get the primitives back from the NURBS tessellator when NURBS property GLU_NURBS_MODE is set to GLU_NURBS_TESSELLATOR (see section 7.6). These callbacks have no effect when GLU_NURBS_MODE is GLU_NURBS_RENDERER.

There are two forms of each callback: one with a pointer to application supplied data and one without. If both versions of a particular callback are specified then the callback with application data will be used. *userData* is specified by calling

void gluNurbsCallbackData(GLUnurbsObj *nurbsObj, void *userData);

The value of *userData* passed to callback functions for a specific NURBS object is the value specified by the last call to **gluNurbsCallbackData**.

All callback functions can be set to NULL even when GLU_NURBS_MODE is set to GLU_NURBS_TESSELLATOR. When a callback function is set to NULL, this callback function will not get invoked and the related data, if any, will be lost.

The **begin** callback indicates the start of a primitive. *type* is one of GL_LINES, GL_LINE_STRIPS, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, GL_TRIANGLES or GL_QUAD_STRIP. The default begin callback function is NULL.

The vertex callback indicates a vertex of the primitive. The coordinates of the vertex are stored in the parameter *vertex*. All the generated vertices have dimension 3; that is, homogeneous coordinates have been transformed into affine coordinates. The default vertex callback function is NULL.

The normal callback is invoked as the vertex normal is generated. The components of the normal are stored in the parameter *normal*. In the case of a NURBS curve, the callback function is effective only when the user provides a normal map (GL_MAP1_NORMAL). In the case of a NURBS surface, if a normal map (GL_MAP2_NORMAL) is provided, then the generated normal is computed from the normal map. If a normal map is not provided then a surface normal is computed in a manner similar to that described for evaluators when GL_AUT0_NORMAL is enabled. The default normal callback function is NULL.

The color callback is invoked as the color of a vertex is generated. The components of the color are stored in the parameter *color*. This callback is effective only when the user provides a color map (GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4). *color* contains four components: R,G,B,A. The default color callback function is NULL.

The texture callback is invoked as the texture coordinates of a vertex are generated. These coordinates are stored in the parameter *tex_coord*. The number of texture coordinates can be 1, 2, 3 or 4 depending on which type of texture map is specified (GL_MAP*_TEXTURE_COORD_1, GL_MAP*_TEXTURE_COORD_2, GL_MAP*_TEXTURE_COORD_3, GL_MAP*_TEXTURE_COORD_4 where * can be either 1 or 2). If no texture map is specified, this callback function will not be called. The default texture callback function is NULL.

The end callback is invoked at the end of a primitive. The default end callback function is NULL.

The error callback is invoked when a NURBS function detects an error condition. There are 37 errors specific to NURBS functions, and they are named GLU_NURBS_ERROR1 through GLU_NURBS_ERROR37. Strings describing the meaning of these error codes can be retrieved with gluErrorString.

7.3 NURBS Curves

NURBS curves are specified with the following routines:

void gluBeginCurve(GLUnurbsObj *nurbsObj);

void gluNurbsCurve(GLUnurbsObj *nurbsObj, GLint nknots, GLfloat *knot, GLint stride, GLfloat *ctlarray, GLint order, GLenum type);

```
void gluEndCurve( GLUnurbsObj *nurbsObj );
```

gluBeginCurve and gluEndCurve delimit a curve definition. After the gluBeginCurve and before the gluEndCurve, a series of gluNurbsCurve calls specify the attributes of the curve. *type* can be any of the one dimensional evaluators (such as GL_MAP1_VERTEX_3). *knot* points to an array of monotonically increasing knot values, and *nknots* tells how many knots are in the array. *ctlarray* points to an array of control points, and *order* indicates the order of the curve. The number of control points in *ctlarray* will be equal to *nknots* - *order*. Lastly, *stride* indicates the offset (expressed in terms of single precision values) between control points.

The NURBS curve attribute definitions must include either a GL_MAP1_VERTEX3 description or a GL_MAP1_VERTEX4 description.

At the point that **gluEndCurve** is called, the curve will be tessellated into line segments and rendered with the aid of OpenGL evaluators. **gl-PushAttrib** and **glPopAttrib** are used to preserve the previous evaluator state during rendering.

7.4 NURBS Surfaces

NURBS surfaces are described with the following routines:

void gluBeginSurface(GLUnurbsObj *nurbsObj);

```
void gluNurbsSurface( GLUnurbsObj *nurbsObj,
GLint sknot_count, GLfloat *sknot, GLint tknot_count,
GLfloat *tknot, GLint s_stride, GLint t_stride,
GLfloat *ctlarray, GLint sorder, GLint torder,
GLenum type );
```

void gluEndSurface(GLUnurbsObj *nurbsObj);

The surface description is almost identical to the curve description. gluBeginSurface and gluEndSurface delimit a surface definition. After the gluBeginSurface, and before the gluEndSurface, a series of gluNurbsSurface calls specify the attributes of the surface. type can be any of the two dimensional evaluators (such as GL_MAP2_VERTEX_3). sknot and tknot point to arrays of monotonically increasing knot values, and sknot_count and tknot_count indicate how many knots are in each array. ctlarray points to an array of control points, and sorder and torder indicate the order of the surface in both the s and t directions. The number of control points in ctlarray will be equal to $(sknot_count - sorder) \times (tknot_count - torder)$. Finally, s_stride and t_stride indicate the offset in single precision values between control points in the s and t directions.

The NURBS surface, like the NURBS curve, must include an attribute definition of type GL_MAP2_VERTEX3 or GL_MAP2_VERTEX4.

When **gluEndSurface** is called, the NURBS surface will be tessellated and rendered with the aid of OpenGL evaluators. The evaluator state is preserved during rendering with **glPushAttrib** and **glPopAttrib**.

7.5 Trimming

A trimming region defines a subset of the NURBS surface domain to be evaluated. By limiting the part of the domain that is evaluated, it is possible to create NURBS surfaces that contain holes or have smooth boundaries.

A trimming region is defined by a set of closed trimming loops in the parameter space of a surface. When a loop is oriented counter-clockwise, the area within the loop is retained, and the part outside is discarded. When the loop is oriented clockwise, the area within the loop is discarded, and the rest is retained. Loops may be nested, but a nested loop must be oriented oppositely from the loop that contains it. The outermost loop must be oriented counter-clockwise.

A trimming loop consists of a connected sequence of NURBS curves and piecewise linear curves. The last point of every curve in the sequence must be the same as the first point of the next curve, and the last point of the last curve must be the same as the first point of the first curve. Self-intersecting curves are not allowed.

To define trimming loops, use the following routines:

```
void gluBeginTrim( GLUnurbsObj *nurbsObj );
void gluPwlCurve( GLUnurbsObj *nurbsObj, GLint count,
    GLfloat *array, GLint stride, GLenum type );
void gluNurbsCurve( GLUnurbsObj *nurbsObj,
    GLint nknots, GLfloat *knot, GLint stride,
    GLfloat *ctlarray, GLint order, GLenum type );
```

```
void gluEndTrim( GLUnurbsObj *nurbsObj );
```

A NURBS trimming curve is very similar to a regular NURBS curve, with the major difference being that a NURBS trimming curve exists in the parameter space of a NURBS surface.

gluPwlCurve defines a piecewise linear curve. *count* indicates how many points are on the curve, and *array* points to an array containing the curve points. *stride* indicates the offset in single precision values between curve points.

type for both gluPwlCurve and gluNurbsCurve can be either GLU_MAP1_TRIM_2 or GLU_MAP1_TRIM_3. GLU_MAP1_TRIM_2 curves define trimming regions in two dimensional (s and t) parameter space. The GLU_MAP1_TRIM_3 curves define trimming regions in two dimensional homogeneous (s, t and q) parameter space.

Note that the trimming loops must be defined at the same time that the surface is defined (between **gluBeginSurface** and **gluEndSurface**).

7.6 NURBS Properties

A set of properties associated with a NURBS object affects the way that NURBS are rendered or tessellated. These properties can be adjusted by the user.

```
void gluNurbsProperty( GLUnurbsObj *nurbsObj,
GLenum property, GLfloat value );
```

allows the user to set one of the following properties: GLU_CULLING, GLU_SAMPLING_TOLERANCE, GLU_SAMPLING_METHOD, GLU_PARAMETRIC_TOLERANCE, GLU_DISPLAY_MODE, GLU_AUTO_LOAD_MATRIX, GLU_U_STEP, GLU_V_STEP and GLU_NURBS_MODE. *property* indicates the property to be modified, and *value* specifies the new value.

GLU_NURBS_MODE should be set to either GLU_NURBS_RENDERER or GLU_NURBS_TESSELLATOR. When set to GLU_NURBS_RENDERER, NURBS objects are tessellated into OpenGL evaluators and sent to the pipeline for rendering. When set to GLU_NURBS_TESSELLATOR, NURBS objects are tessellated into a sequence of primitives such as lines, triangles and triangle strips, but the vertices, normals, colors, and/or textures are retrieved back through a callback interface as specified in Section 7.2. This allows the user to cache the tessellated results for further processing. The default value is GLU_NURBS_RENDERER

The GLU_CULLING property is a boolean value (*value* should be set to either GL_TRUE or GL_FALSE). When set to GL_TRUE, it indicates that a NURBS curve or surface should be discarded prior to tessellation if its control polyhedron lies outside the current viewport. The default is GL_FALSE.

GLU_SAMPLING_METHOD specifies how NURBS а surface should value may be set to one of GLU_PATH_LENGTH, be tessellated. GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_OBJECT_PATH_LENGTH GLU_OBJECT_PARAMETRIC_ERROR. When set to GLU_PATH_LENGTH, the surface is rendered so that the maximum length, in pixels, of the edges of the tessellation polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE. GLU_PARAMETRIC_ERROR specifies that the surface is rendered in such a way that the value specified by GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate. GLU_DOMAIN_DISTANCE allows users to specify, in parametric coordinates, how many sample points per unit length are taken in u, v dimension. GLU_OBJECT_PATH_LENGTH is similar to GLU_PATH_LENGTH except that it is view independent; that is, it specifies that the surface is rendered so that the maximum length, in object space, of edges of the tessellation polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE. GLU_OBJECT_PARAMETRIC_ERROR is similar to GLU_PARAMETRIC_ERROR except that the surface is rendered in such a way that the value specified by GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in object space, between the tessellation polygons and the surfaces they approximate. The default value of GLU_SAMPLING_METHOD is GLU_PATH_LENGTH.

GLU_SAMPLING_TOLERANCE specifies the maximum length, in pixels or in object space length unit, to use when the sampling method is set to GLU_PATH_LENGTH or GLU_OBJECT_PATH_LENGTH. The default value is 50.0.

GLU_PARAMETRIC_TOLERANCE specifies the maximum distance, in pixels or in object space length unit, to use when the sampling method is set to GLU_PARAMETRIC_ERROR or GLU_OBJECT_PARAMETRIC_ERROR. The default value for GLU_PARAMETRIC_TOLERANCE is 0.5.

GLU_JSTEP specifies the number of sample points per unit length taken along the u dimension in parametric coordinates. It is needed when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The default value is 100.

GLU_V_STEP specifies the number of sample points per unit length taken along the v dimension in parametric coordinates. It is needed when GLU_SAMPLING_METHOD is set to GLU_DOMAIN_DISTANCE. The default value is 100.

GLU_AUTO_LOAD_MATRIX is a boolean value. When it is set to GL_TRUE, the NURBS code will download the projection matrix, the model view matrix, and the viewport from the OpenGL server in order to compute sampling and culling matrices for each curve or surface that is rendered. These matrices are required to tessellate a curve or surface and to cull it if it lies outside the viewport. If this mode is turned off, then the user needs to provide a projection matrix, a model view matrix, and a viewport that the NURBS code can use to construct sampling and culling matrices. This can be done with the gluLoadSamplingMatrices function:

Until the GLU_AUTO_LOAD_MATRIX property is turned back on, the NURBS routines will continue to use whatever sampling and culling matrices are stored in the NURBS object. The default for GLU_AUTO_LOAD_MATRIX is GL_TRUE.

You may get unexpected results when GLU_AUTO_LOAD_MATRIX is enabled and the results of the NURBS tesselation are being stored in a display list, since the OpenGL matrices which are used to create the sampling and culling matrices will be those that are in effect when the list is created, not those in effect when it is executed.

GLU_DISPLAY_MODE specifies how a NURBS surface should be rendered. value may be set to one of GLU_FILL, GLU_OUTLINE_POLY or GLU_OUTLINE_PATCH. When GLU_NURBS_MODE is set to be GLU_NURBS_RENDERER, value defines how a NURBS surface should be rendered. When set to GLU_FILL, the surface is rendered as a set of polygons. GLU_OUTLINE_POLY instructs the NURBS library to draw only the outlines of the polygons created by tessellation. GLU_OUTLINE_PATCH will cause just the outlines of patches and trim curves defined by the user to be drawn. When GLU_NURBS_MODE is set to be GLU_NURBS_TESSELLATOR, value defines how a NURBS surface should be tessellated. When GLU_DISPLAY_MODE is set to GLU_FILL or GLU_OUTLINE_POLY, the NURBS surface is tessellated into OpenGL triangle primitives which can be retrieved back through callback functions. If value is set to GLU_OUTLINE_PATCH, only the outlines of the patches and trim curves are generated as a sequence of line strips and can be retrieved back through callback functions. The default is GLU_FILL.

Property values can be queried by calling

void gluGetNurbsProperty(GLUnurbsObj *nurbsObj, GLenum property, GLfloat *value);

The specified *property* is returned in *value*.

Chapter 8

Errors

Calling

const GLubyte *gluErrorString(GLenum errorCode);

produces an error string corresponding to a GL or GLU error code. The error string is in ISO Latin 1 format. The standard GLU error codes are GLU_INVALID_ENUM, GLU_INVALID_VALUE, GLU_INVALID_OPERATION and GLU_OUT_OF_MEMORY. There are also specific error codes for polygon tessellation, quadrics, and NURBS as described in their respective sections.

If an invalid call to the underlying OpenGL implementation is made by GLU, either GLU or OpenGL errors may be generated, depending on where the error is detected. This condition may occur only when making a GLU call introduced in a later version of GLU than that corresponding to the OpenGL implementation (see Chapter 9); for example, calling gluBuild3DMipmaps or passing packed pixel types to gluScaleImage when the underlying OpenGL version is earlier than 1.2.

Chapter 9

GLU Versions

Each version of GLU corresponds to the OpenGL version shown in Table 9.1; GLU features introduced in a particular version of GLU may not be usable if the underlying OpenGL implementation is an earlier version.

All versions of GLU are upward compatible with earlier versions, meaning that any program that runs with the earlier implementation will run unchanged with any later GLU implementation.

9.1 GLU 1.1

In GLU 1.1, gluGetString was added allowing the GLU version number and GLU extensions to be queried. Also, the NURBS properties GLU_SAMPLING_METHOD, GLU_PARAMETRIC_TOLERANCE, GLU_U_STEP and GLU_V_STEP were added providing support for different tesselation methods. In GLU 1.0, the only sampling method supported was GLU_PATH_LENGTH.

GLU Version	Corresponding OpenGL Version
GLU 1.0	OpenGL 1.0
GLU 1.1	OpenGL 1.0
GLU 1.2	OpenGL 1.1
GLU 1.3	OpenGL 1.2

Table 9.1: Relationship of OpenGL and GLU versions.

9.2 GLU 1.2

A new polygon tesselation interface was added in GLU 1.2. See section 5.7 for more information on the API changes.

A new NURBS callback interface and object space sampling methods was also added in GLU 1.2. See sections 7.2 and 7.6 for API changes.

9.3 GLU 1.3

The **gluCheckExtension** utility function was introduced.

gluScaleImage and gluBuild*x*DMipmaps support the new packed pixel formats and types introduced by OpenGL 1.2.

gluBuild3DMipmaps was added to support 3D textures, introduced by OpenGL 1.2.

gluBuild*x*DMipmapLevels was added to support OpenGL 1.2's ability to load only a subset of mipmap levels.

gluUnproject4 was added for use when non-default depth range or w values other than 1 need to be specified.

New gluNurbsCallback callbacks and the GLU_NURBS_MODE NURBS property were introduced to allow applications to capture NURBS tesselations. These features exactly match corresponding features of the GLU_EXT_nurbs_tessellator GLU extension, and may be used interchange-ably with the extension.

New values of the GLU_SAMPLING_METHOD NURBS property were introduced to support object-space sampling criteria. These features exactly match corresponding features of the GLU_EXT_object_space_tess GLU extension, and may be used interchangeably with the extension.

Index of GLU Commands

beginData, 12, 25 color. 25 colorData, 25 combine, 12 combineData, 12 edgeFlag, 12 edgeFlagData, 12 end, 12, 25 endData, 12, 25 error, 12, 25 errorData, 12 GL_4D_COLOR_TEXTURE, 9 GL_AUTO_NORMAL, 26 GL_FALSE, 2, 9, 13, 15, 21, 30 GL_LINE_LOOP, 15 GL_LINE_STRIPS, 26 GL_LINES, 26 GL_MAP*_TEXTURE_COORD_1, 26GL_MAP*_TEXTURE_COORD_2, 26GL_MAP*_TEXTURE_COORD_3, 26GL_MAP*_TEXTURE_COORD_4, 26GL_MAP1_COLOR_4, 26 GL_MAP1_NORMAL, 26 GL_MAP1_VERTEX3, 27 GL_MAP1_VERTEX4, 27 GL_MAP1_VERTEX_3, 27 GL_MAP2_COLOR_4, 26 GL_MAP2_NORMAL, 26 GL_MAP2_VERTEX3, 28

begin, 12, 25

GL_MAP2_VERTEX4, 28 GL_MAP2_VERTEX_3, 28 GL_QUAD_STRIP, 26 GL_TRIANGLE_FAN,13, 26 GL_TRIANGLE_STRIP,13, 26 GL_TRIANGLES,13, 26 GL_TRUE, 2, 9, 13, 15, 21, 30, 31 GL_VIEWPORT, 8 glBegin, 13 glDepthRange, 9 glDrawPixels, 5 glFeedbackBuffer, 9 glGetDoublev, 9 glGetIntegerv, 8, 9 glGetString, 3 glMultMatrix, 7 glNewList, 1 glOrtho, 7 glPopAttrib, 27, 28 glPushAttrib, 27, 28 glTexImage1D, 5 glTexImage2D, 5 glTexImage3D, 5 glTexImagexD, 6 GLU_AUTO_LOAD_MATRIX, 30, 31 GLU_BEGIN, 19 GLU_CCW, 18 GLU_CULLING, 30 GLU_CW, 18 GLU_DISPLAY_MODE, 30-32 GLU_DOMAIN_DISTANCE, 30, 31 GLU_EDGE_FLAG, 19 GLU_END, 19, 25 GLU_END_DATA, 25 GLU_ERROR, 19, 21, 25 GLU_EXTENSIONS, 2

36

INDEX

GLU_EXTERIOR, 18 GLU_FILL, 22, 23, 31, 32 GLU_FLAT, 21 GLU_INSIDE, 21 GLU_INTERIOR, 18 GLU_INVALID_ENUM, 33 GLU_INVALID_OPERATION, 33 GLU_INVALID_VALUE, 6, 33 GLU_LINE, 22 GLU_MAP1_TRIM_2, 29 GLU_MAP1_TRIM_3, 29 GLU_NONE, 21 GLU_NORMAL, 25 GLU_NORMAL_DATA, 25 GLU_NURBS_BEGIN, 25 GLU_NURBS_BEGIN_DATA, 25 GLU_NURBS_COLOR, 25 GLU_NURBS_COLOR_DATA, 25 GLU_NURBS_ERROR1, 27 GLU_NURBS_ERROR37, 27 GLU_NURBS_MODE, 25, 26, 30--32, 35GLU_NURBS_RENDERER, 25, 30, 31GLU_NURBS_TESSELLATOR, 25, 26,30, 32 GLU_NURBS_TEXTURE_COORD, 25GLU_NURBS_TEXTURE_COORD_ DATA. 25 GLU_NURBS_VERTEX, 25 GLU_NURBS_VERTEX_DATA, 25 GLU_OBJECT_PARAMETRIC_ ERROR, 30, 31 GLU_OBJECT_PATH_LENGTH, 30, 31GLU_OUT_OF_MEMORY, 33 GLU_OUTLINE_PATCH, 31, 32 GLU_OUTLINE_POLY,31, 32 GLU_OUTSIDE,21, 22 GLU_PARAMETRIC_ERROR,30, 31GLU_PARAMETRIC_ TOLERANCE, 30, 31, 34 GLU_PATH_LENGTH, 30, 31, 34

GLU_POINT, 22 GLU_SAMPLING_METHOD, 30, 31, 34, 35 GLU_SAMPLING_TOLERANCE, 30GLU_SILHOUETTE, 22, 23 GLU_SMOOTH, 21 GLU_TESS_BEGIN, 12, 15, 19 GLU_TESS_BEGIN_DATA, 12, 15 GLU_TESS_BOUNDARY_ONLY,13, 15, 17GLU_TESS_COMBINE, 12 GLU_TESS_COMBINE_DATA, 12 GLU_TESS_COORD_TOO_LARGE, 13GLU_TESS_EDGE_FLAG, 12, 19 GLU_TESS_EDGE_FLAG_DATA, 12 GLU_TESS_END,12, 19 GLU_TESS_END_DATA, 12 GLU_TESS_ERROR, 12, 19 GLU_TESS_ERROR_DATA, 12 GLU_TESS_MAX_COORD_TOO_ LARGE, 13 GLU_TESS_MISSING_BEGIN_ CONTOUR, 13 GLU_TESS_MISSING_BEGIN_ POLYGON, 13 GLU_TESS_MISSING_END_ CONTOUR, 13 GLU_TESS_MISSING_END_ POLYGON, 13 GLU_TESS_NEED_COMBINE_ CALLBACK, 13, 14 GLU_TESS_TOLERANCE, 15 GLU_TESS_TOLERANCE., 15 GLU_TESS_VERTEX,12, 19 GLU_TESS_VERTEX_DATA, 12 GLU_TESS_WINDING_ABS_GEQ_ TWO,15, 17 GLU_TESS_WINDING_NEGATIVE, 15GLU_TESS_WINDING_NONZERO, 15, 17 GLU_TESS_WINDING_ODD, 15 GLU_TESS_WINDING_POSITIVE,

15, 17 GLU_TESS_WINDING_RULE, 15 GLU_U_STEP, 30, 31, 34 GLU_UNKNOWN, 18 GLU_V_STEP, 30, 31, 34 GLU_VERSION, 2 GLU_VERTEX, 19 gluBeginCurve, 27 gluBeginPolygon, 18, 19 gluBeginSurface, 27–29 gluBeginTrim, 29 gluBuild1DMipmapLevels, 5 gluBuild1DMipmaps, 5 gluBuild2DMipmapLevels, 6 gluBuild2DMipmaps, 5 gluBuild3DMipmapLevels, 6 gluBuild3DMipmaps, 5, 33, 35 gluBuildxDMipmapLevels, 35 gluBuild*x*DMipmaps, 6, 35 gluCheckExtension, 2, 3, 35 gluCylinder, 22 gluDeleteNurbsRenderer, 24, 25 gluDeleteQuadric, 20 gluDeleteTess, 10 gluDisk, 23 gluEndCurve, 27 gluEndPolygon, 18, 19 gluEndSurface, 28, 29 gluEndTrim, 29 gluErrorString, 5, 21, 27, 33 gluGetNurbsProperty, 32 gluGetString, 2, 3, 34 gluGetTessProperty, 16 gluLoadSamplingMatrices, 31 gluLookAt, 8 gluNewNurbsRenderer, 24 gluNewQuadric, 20 gluNewTess, 10 gluNextContour, 18, 19 gluNurbsCallback, 25, 35 gluNurbsCallbackData, 26 gluNurbsCurve, 27, 29 gluNurbsProperty, 29 gluNurbsSurface, 28 gluOrtho2D, 7

gluPartialDisk, 23 gluPerspective, 7 gluPickMatrix, 8 gluProject, 9 gluPwlCurve, 29 gluQuadricCallback, 20, 21 gluQuadricDrawStyle, 22 gluQuadricNormals, 21 gluQuadricOrientation, 21 gluQuadricTexture, 21 gluScaleImage, 4, 5, 33, 35 gluSphere, 22 gluTessBeginContour, 11, 19 gluTessBeginPolygon, 11, 13, 19 gluTessCallback, 12 gluTessEndContour, 11, 19 gluTessEndPolygon, 11, 19 gluTessNormal, 16–18 gluTessProperty, 14 gluTessVertex, 11, 13 gluUnProject, 9 gluUnProject4, 9 gluUnproject4, 35 glXGetClientString, 3 normal, 25 normalData, 25 texCoord, 25 texCoordData, 25 vertex, 12, 25 vertexData, 12, 25

The OpenGL Utility Toolkit (GLUT) Programming Interface

API Version 3

Mark J. Kilgard Silicon Graphics, Inc.

November 13, 1996

OpenGL is a trademark of Silicon Graphics, Inc. X Window System is a trademark of X Consortium, Inc. Spaceball is a registered trademark of Spatial Systems Inc.

The author has taken care in preparation of this documentation but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising from the use of information or programs contained herein.

Copyright © 1994, 1995, 1996. Mark J. Kilgard. All rights reserved.

All rights reserved. No part of this documentation may be reproduced, in any form or by any means, without permission in writing from the author.

Contents

1	Intro	duction
	1.1	Background
	1.2	Design Philosophy
	1.3	API Version 2
	1.4	API Version 3
	1.5	Conventions
	1.6	Terminology
•	- • •	
2		alization
	2.1	glutInit
	2.2	glutInitWindowPosition, glutInitWindowSize
	2.3	glutInitDisplayMode
3	Begi	nning Event Processing
	3.1	glutMainLoop
4		low Management
	4.1	glutCreateWindow
	4.2	glutCreateSubWindow
	4.3	glutSetWindow, glutGetWindow 10
	4.4	glutDestroyWindow
	4.5	glutPostRedisplay
	4.6	glutSwapBuffers
	4.7	glutPositionWindow
	4.8	glutReshapeWindow
	4.9	glutFullScreen
	4.10	glutPopWindow, glutPushWindow
	4.11	glutShowWindow, glutHideWindow, glutIconifyWindow
	4.12	glutSetWindowTitle, glutSetIconTitle
	4.13	glutSetCursor
5	Over	lay Management 14
5	5.1	glutEstablishOverlay
	5.2	glutustabilisiovenay
		glutoschayer i glutos
	5.4	glutPostOverlayRedisplay
	5. 4	glut ostovenay keuispay glut Blow Overlay, glut Hide Overlay from the second se
	0.0	
6	Men	u Management 10
	6.1	glutCreateMenu
	6.2	glutSetMenu, glutGetMenu 17
	6.3	glutDestroyMenu
	6.4	glutAddMenuEntry
	6.5	glutAddSubMenu
	6.6	glutChangeToMenuEntry 18
	6.7	glutChangeToSubMenu
	6.8	glutRemoveMenuItem
	6.9	glutAttachMenu, glutDetachMenu

stration				
ayFunc				

7	Call	back Registration	19
	7.1	glutDisplayFunc	20
	7.2	glutOverlayDisplayFunc	20
	7.3	glutReshapeFunc	21
	7.4	glutKeyboardFunc	21
	7.5	glutMouseFunc	22
	7.6	glutMotionFunc, glutPassiveMotionFunc	22
	7.7	glutVisibilityFunc	23
	7.8	glutEntryFunc	23
	7.9	glutSpecialFunc	24
	7.10	glutSpaceballMotionFunc	24
	7.11	glutSpaceballRotateFunc	25
		glutSpaceballButtonFunc	25
	7.13	glutButtonBoxFunc	26
	7.14	glutDialsFunc	26
	7.15	glutTabletMotionFunc	27
	7.16	glutTabletButtonFunc	27
		glutMenuStatusFunc	27
	7.18	glutIdleFunc	28
		glutTimerFunc	28
		-	
8		r Index Colormap Management	29
	8.1	glutSetColor	29
	8.2	glutGetColor	29
	8.3	glutCopyColormap	30
9	State	Retrieval	30
-	9.1	glutGet	30
	9.2	glutLayerGet	32
	9.3	glutDeviceGet	32
	9.4	glutGetModifiers	33
	9.5	glutextensionSupported	33
	2.0		55
10	Font	Rendering	34
	10.1	glutBitmapCharacter	34
	10.2	glutBitmapWidth	35
	10.3	glutStrokeCharacter	35
	10.4	glutStrokeWidth	36
11	Coor	notaio Obiost Doudoning	36
11		netric Object Rendering	3 6
		glutSolidSphere, glutWireSphere	
		glutSolidCube, glutWireCube	36
		glutSolidCone, glutWireCone	37
		glutSolidTorus, glutWireTorus	37
		glutSolidDodecahedron, glutWireDodecahedron	38
		glutSolidOctahedron, glutWireOctahedron	38
		glutSolidTetrahedron, glutWireTetrahedron	38
		glutSolidIcosahedron, glutWireIcosahedron	38
	11.9	glutSolidTeapot, glutWireTeapot	39

13	FORTRAN Binding	41					
	13.1 Names for the FORTRAN GLUT Binding	41					
	13.2 Font Naming Caveat						
	13.3 NULL Callback	42					
14	Implementation Issues	42					
	14.1 Name Space Conventions	42					
	14.2 Modular Implementation						
	14.3 Error Checking and Reporting						
	14.4 Avoid Unspecified GLUT Usage Restrictions	42					
A	GLUT State	44					
	A.1 Types of State	44					
	A.2 Global State						
	A.3 Window State	45					
	A.4 Menu State	48					
B	glut.h ANSI C Header File	49					
С	fglut.h FORTRAN Header File	55					
Re	References						
Inc	Index						

1 Introduction

The OpenGL Utility Toolkit (GLUT) is a programming interface with ANSI C and FORTRAN bindings for writing window system independent OpenGL programs. The toolkit supports the following functionality:

- Multiple windows for OpenGL rendering.
- Callback driven event processing.
- Sophisticated input devices.
- An "idle" routine and timers.
- A simple, cascading pop-up menu facility.
- Utility routines to generate various solid and wire frame objects.
- Support for bitmap and stroke fonts.
- Miscellaneous window management functions, including managing overlays.

An ANSI C implementation of GLUT for the X Window System [15] has been implemented by the author. Windows NT and OS/2 versions of GLUT are also available.

This documentation serves as both a specification and a programming guide. If you are interested in a brief introduction to programming with GLUT, look for the introductory OpenGL column [9] published in *The X Journal*. For a complete introduction to using GLUT, obtain the book *Programming OpenGL for the X Window System* [10]. GLUT is also used by the 2nd edition of the *OpenGL Programming Guide*. Teachers and students interested in using GLUT in conjunction with a college-level computer graphics class should investigate Angel's textbook *Interactive Computer Graphics: A top-down approach with OpenGL* [2] that uses GLUT for its OpenGL-based examples programs.

The remainder of this section describes GLUT's design philosophy and usage model. The following sections specify the GLUT routines, grouped by functionality. The final sections discuss usage advice, the FORTRAN binding, and implementation issues. Appendix A enumerates and annotates the logical programmer visible state maintained by GLUT. Appendix B presents the ANSI C GLUT API via its header file. Appendix C presents the FORTRAN GLUT API via its header file.

1.1 Background

One of the major accomplishments in the specification of OpenGL [16, 12] was the isolation of window system dependencies from OpenGL's rendering model. The result is that OpenGL is window system independent.

Window system operations such as the creation of a rendering window and the handling of window system events are left to the native window system to define. Necessary interactions between OpenGL and the window system such as creating and binding an OpenGL context to a window are described separately from the OpenGL specification in a window system dependent specification. For example, the GLX specification [4] describes the standard by which OpenGL interacts with the X Window System.

The predecessor to OpenGL is IRIS GL [17, 18]. Unlike OpenGL, IRIS GL *does* specify how rendering windows are created and manipulated. IRIS GL's windowing interface is reasonably popular largely because it is simple to use. IRIS GL programmers can worry about graphics programming without needing to be an expert in programming the native window system. Experience also demonstrated that IRIS GL's windowing interface was high-level enough that it could be retargeted to different window systems. Silicon Graphics migrated from NeWS to the X Window System without any major changes to IRIS GL's basic windowing interface.

Removing window system operations from OpenGL is a sound decision because it allows the OpenGL graphics system to be retargeted to various systems including powerful but expensive graphics workstations as well as mass-production graphics systems like video games, set-top boxes for interactive television, and PCs.

Unfortunately, the lack of a window system interface for OpenGL is a gap in OpenGL's utility. Learning native window system APIs such as the X Window System's Xlib [7] or Motif [8] can be daunting. Even those familiar with native window system APIs need to understand the interface that binds OpenGL to the native

window system. And when an OpenGL program is written using the native window system interface, despite the portability of the program's OpenGL rendering code, the program itself will be window system dependent.

Testing and documenting OpenGL's functionality lead to the development of the tk and aux toolkits. The aux toolkit is used in the examples found in the *OpenGL Programming Guide* [11]. Unfortunately, aux has numerous limitations and its utility is largely limited to toy programs. The tk library has more functionality than aux but was developed in an *ad hoc* fashion and still lacks much important functionality that IRIS GL programmers expect, like pop-up menus and overlays.

GLUT is designed to fill the need for a window system independent programming interface for OpenGL programs. The interface is designed to be simple yet still meet the needs of useful OpenGL programs. Features from the IRIS GL, aux, and tk interfaces are included to make it easy for programmers used to these interfaces to develop programs for GLUT.

1.2 Design Philosophy

GLUT simplifies the implementation of programs using OpenGL rendering. The GLUT application programming interface (API) requires very few routines to display a graphics scene rendered using OpenGL. The GLUT API (like the OpenGL API) is stateful. Most initial GLUT state is defined and the initial state is reasonable for simple programs.

The GLUT routines also take relatively few parameters. No pointers are returned. The only pointers passed into GLUT are pointers to character strings (all strings passed to GLUT are copied, not referenced) and opaque font handles.

The GLUT API is (as much as reasonable) window system independent. For this reason, GLUT does not return *any* native window system handles, pointers, or other data structures. More subtle window system dependencies such as reliance on window system dependent fonts are avoided by GLUT; instead, GLUT supplies its own (limited) set of fonts.

For programming ease, GLUT provides a simple menu sub-API. While the menuing support is designed to be implemented as pop-up menus, GLUT gives window system leeway to support the menu functionality in another manner (pull-down menus for example).

Two of the most important pieces of GLUT state are the *current window* and *current menu*. Most window and menu routines affect the *current window* or *menu* respectively. Most callbacks implicitly set the *current window* and *menu* to the appropriate window or menu responsible for the callback. GLUT is designed so that a program with only a single window and/or menu will not need to keep track of any window or menu identifiers. This greatly simplifies very simple GLUT programs.

GLUT is designed for simple to moderately complex programs focused on OpenGL rendering. GLUT implements its own event loop. For this reason, mixing GLUT with other APIs that demand their own event handling structure may be difficult. The advantage of a builtin event dispatch loop is simplicity.

GLUT contains routines for rendering fonts and geometric objects, however GLUT makes no claims on the OpenGL display list name space. For this reason, none of the GLUT rendering routines use OpenGL display lists. It is up to the GLUT programmer to compile the output from GLUT rendering routines into display lists if this is desired.

GLUT routines are logically organized into several sub-APIs according to their functionality. The sub-APIs are:

Initialization. Command line processing, window system initialization, and initial window creation state are controlled by these routines.

Beginning Event Processing. This routine enters GLUT's event processing loop. This routine never returns, and it continuously calls GLUT callbacks as necessary.

Window Management. These routines create and control windows.

Overlay Management. These routines establish and manage overlays for windows.

Menu Management. These routines create and control pop-up menus.

Callback Registration. These routines register callbacks to be called by the GLUT event processing loop.

Color Index Colormap Management. These routines allow the manipulation of color index colormaps for windows.

State Retrieval. These routines allows programs to retrieve state from GLUT.

Font Rendering. These routines allow rendering of stroke and bitmap fonts.

Geometric Shape Rendering. These routines allow the rendering of 3D geometric objects including spheres, cones, icosahedrons, and teapots.

1.3 API Version 2

In response to feedback from the original version of GLUT, GLUT API version 2 was developed. Additions to the original GLUT API version 1 are:

- Support for requesting stereo and multisample windows.
- New routines to query support for and provide callbacks for sophisticated input devices: the Spaceball, tablet, and dial & button box.
- New routine to register a callback for keyboard function and directional keys. In version 1, only ASCII characters could be generated.
- New queries for stereo, multisampling, and elapsed time.
- New routine to ease querying for OpenGL extension support.

GLUT API version 2 is completely compatible with version 1 of the API.

1.4 API Version 3

Further feedback lead to the development of GLUT API version 3. Additions to the GLUT API version 2 are:

- The glutMenuStateFunc has been deprecated in favor of the glutMenuStatusFunc.
- glutFullScreen requests full screen top-level windows.
- Three additional Helvetica bitmap fonts.
- Implementations should enforce not allowing any modifications to menus while menus are in use.
- glutBitmapWidth and glutStrokeBitmap return the widths of individual characters.
- glutGetModifiers called during a keyboard, mouse, or special callback returns the modifiers (Shift, Ctrl, Alt) held down when the mouse or keyboard event was generated.
- Access to per-window transparent overlays when overlay hardware is supported. The routines added are glutEstablishOverlay, glutRemoveOverlay, glutShowOverlay, glutHideOverlay, glutUseOverlay, glutLayerGet, and glutPostOverlayRedisplay.
- A new display mode called GLUT_LUMINANCE using OpenGL's RGBA color model, but that has no green or blue components. The red component is converted to an index and looked up in a writable colormap to determine displayed colors. See glutInitDisplayMode.

GLUT API version 3 should be largely compatible with version 2. Be aware that programs that used to (through some degree of fortuitous timing) modify menus while menus are in use will encounter fatal errors when doing so in version 3.

Another change in GLUT 3.0 that may require source code modification to pre-3.0 GLUT programs. GLUT 3.0 no longer lets a window be shown without a display callback registered. This change makes sure windows are not displayed on the screen without the GLUT application providing a way for them to be rendered. In

conjunction with this change, glutDisplayFunc no longer allows NULL to deregister a display callback. While there is no longer a way to deregister a display callback, you can still change the change the display callback routine with subsequent calls to glutDisplayFunc.

The display mode mask parameter for glutInitDisplayMode and the milliseconds parameter for glutTimerFunc are now of type unsigned int (previously unsigned long).

1.5 Conventions

GLUT window and screen coordinates are expressed in pixels. The upper left hand corner of the screen or a window is (0,0). X coordinates increase in a rightward direction; Y coordinates increase in a downward direction. Note: This is inconsistent with OpenGL's coordinate scheme that generally considers the lower left hand coordinate of a window to be at (0,0) but is consistent with most popular window systems.

Integer identifiers in GLUT begin with one, not zero. So window identifiers, menu identifiers, and menu item indices are based from one, not zero.

In GLUT's ANSIC binding, for most routines, basic types (int, char*) are used as parameters. In routines where the parameters are directly passed to OpenGL routines, OpenGL types (GLfloat) are used.

The header files for GLUT should be included in GLUT programs with the following include directive:

#include <GL/glut.h>

Because a very large window system software vendor (who will remain nameless) has an apparent inability to appreciate that OpenGL's API is independent of their window system API, portable ANSI C GLUT programs should not directly include <GL/gl.h> or <GL/glu.h>. Instead, ANSI C GLUT programs should rely on <GL/glut.h> to include the necessary OpenGL and GLU related header files.

The ANSI C GLUT library archive is typically named libglut. a on Unix systems. GLUT programs need to link with the system's OpenGL and GLUT libraries (and any libraries these libraries potentially depend on). A set of window system dependent libraries may also be necessary for linking GLUT programs. For example, programs using the X11 GLUT implementation typically need to link with Xlib, the X extension library, possibly the X Input extension library, the X miscellaneous utilities library, and the math library. An example X11/Unix compile line would look like:

cc -o foo foo.c -lglut -lGLU -lGL -lXmu -lXi -lXext -lX11 -lm

1.6 Terminology

A number of terms are used in a GLUT-specific manner throughout this document. The GLUT meaning of these terms is independent of the window system GLUT is used with. Here are GLUT-specific meanings for the following GLUT-specific terms:

Callback A programmer specified routine that can be registered with GLUT to be called in response to a specific type of event. Also used to refer to a specific callback routine being called.

Colormap A mapping of pixel values to RGB color values. Use by color index windows.

Dials and button box A sophisticated input device consisting of a pad of buttons and an array of rotating dials, often used by computer-aided design programs.

Display mode A set of OpenGL frame buffer capabilities that can be attributed to a window.

- *Idle* A state when no window system events are received for processing as callbacks and the idle callback, if one is registered, is called.
- *Layer in use* Either the normal plane or overlay. This per-window state determines what frame buffer layer OpenGL commands affect.
- Menu entry A menu item that the user can select to trigger the menu callback for the menu entry's value.

Menu item Either a menu entry or a sub-menu trigger.

1.6 Terminology

- *Modifiers* The Shift, Ctrl, and Alt keys that can be held down simultaneously with a key or mouse button being pressed or released.
- *Multisampling* A technique for hardware antialiasing generally available only on expensive 3D graphics hardware [1]. Each pixel is composed of a number of samples (each containing color and depth information). The samples are averaged to determine the displayed pixel color value. Multisampling is supported as an extension to OpenGL.
- Normal plane The default frame buffer layer where GLUT window state resides; as opposed to the overlay.
- **Overlay** A frame buffer layer that can be displayed preferentially to the *normal plane* and supports transparency to display through to the *normal plane*. Overlays are useful for rubber-banding effects, text annotation, and other operations, to avoid damaging the normal plane frame buffer state. Overlays require hardware support not present on all systems.
- *Pop* The act of forcing a window to the top of the stacking order for sibling windows.
- *Pop-up menu* A menu that can be set to appear when a specified mouse button is pressed in a window. A popmenu consists of multiple menu items.
- **Push** The act of forcing a window to the bottom of the stacking order for sibling windows.
- **Reshape** The act of changing the size or shape of the window.
- *Spaceball* A sophisticated 3D input device that provides six degrees of freedom, three axes of rotation and three axes of translation. It also supports a number of buttons. The device is a hand-sized ball attached to a base. By cupping the ball with one's hand and applying torsional or directional force on the ball, rotations and translations are generated.
- *Stereo* A frame buffer capability providing left and right color buffers for creating stereoscopic renderings. Typically, the user wears LCD shuttered goggles synchronized with the alternating display on the screen of the left and right color buffers.
- Sub-menu A menu cascaded from some sub-menu trigger.
- Sub-menu trigger A menu item that the user can enter to cascade another pop-up menu.
- *Subwindow* A type of window that is the child window of a top-level window or other subwindow. The drawing and visible region of a subwindow is limited by its parent window.
- *Tablet* A precise 2D input device. Like a mouse, 2D coordinates are returned. The absolute position of the tablet "puck" on the tablet is returned. Tablets also support a number of buttons.
- *Timer* A callback that can be scheduled to be called in a specified interval of time.
- *Top-level window* A window that can be placed, moved, resized, etc. independently from other top-level windows by the user. Subwindows may reside within a top-level window.
- Window A rectangular area for OpenGL rendering.
- *Window display state* One of shown, hidden, or iconified. A shown window is potentially visible on the screen (it may be obscured by other windows and not actually visible). A hidden window will never be visible. An iconified window is not visible but could be made visible in response to some user action like clicking on the window's corresponding icon.
- *Window system* A broad notion that refers to both the mechanism and policy of the window system. For example, in the X Window System both the window manager and the X server are integral to what GLUT considers the window system.

2 Initialization

Routines beginning with the glutInit- prefix are used to initialize GLUT state. The primary initialization routine is glutInit that should only be called exactly once in a GLUT program. No non-glutInit- pre-fixed GLUT or OpenGL routines should be called before glutInit.

The other glutInit-routines may be called before glutInit. The reason is these routines can be used to set default window initialization state that might be modified by the command processing done in glutInit. For example, glutInitWindowSize(400, 400) can be called before glutInit to indicate 400 by 400 is the program's default window size. Setting the *initial window size* or *position* before glutInit allows the GLUT program user to specify the initial size or position using command line arguments.

2.1 glutInit

glutInit is used to initialize the GLUT library.

Usage

void glutInit(int *argcp, char **argv);

- argcp A pointer to the program's *unmodified* argc variable from main. Upon return, the value pointed to by argcp will be updated, because glutInit extracts any command line options intended for the GLUT library.
- argv The program's *unmodified* argv variable from main. Like argcp, the data for argv will be updated because glutInit extracts any command line options understood by the GLUT library.

Description

glutInit will initialize the GLUT library and negotiate a session with the window system. During this process, glutInit may cause the termination of the GLUT program with an error message to the user if GLUT cannot be properly initialized. Examples of this situation include the failure to connect to the window system, the lack of window system support for OpenGL, and invalid command line options.

glutInit also processes command line options, but the specific options parse are window system dependent.

X Implementation Notes

The X Window System specific options parsed by glutInit are as follows:

- -display **DISPLAY** Specify the X server to connect to. If not specified, the value of the DISPLAY environment variable is used.
- -geometry $W \ge H + X + Y$ Determines where window's should be created on the screen. The parameter following -geometry should be formatted as a standard X geometry specification. The effect of using this option is to change the GLUT *initial size* and *initial position* the same as if glutInitWindowSize or glutInitWindowPosition were called directly.
- -iconic Requests all top-level windows be created in an iconic state.
- -indirect Force the use of *indirect* OpenGL rendering contexts.
- -direct Force the use of *direct* OpenGL rendering contexts (not all GLX implementations support direct rendering contexts). A fatal error is generated if direct rendering is not supported by the OpenGL implementation.

If neither -indirect or -direct are used to force a particular behavior, GLUT will attempt to use direct rendering if possible and otherwise fallback to indirect rendering.

2.2 glutInitWindowPosition, glutInitWindowSize

- -gldebug After processing callbacks and/or events, check if there are any OpenGL errors by calling glGetError. If an error is reported, print out a warning by looking up the error code with gluErrorString. Using this option is helpful in detecting OpenGL run-time errors.
- -sync Enable synchronous X protocol transactions. This option makes it easier to track down potential X protocol errors.

2.2 glutInitWindowPosition, glutInitWindowSize

glutInitWindowPosition and glutInitWindowSize set the *initial window position* and *size* respectively.

Usage

```
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
```

width Width in pixels.

height Height in pixels.

x Window X location in pixels.

y Window Y location in pixels.

Description

Windows created by glutCreateWindow will be requested to be created with the current *initial window position* and *size*.

The initial value of the *initial window position* GLUT state is -1 and -1. If either the X or Y component to the *initial window position* is negative, the actual window position is left to the window system to determine. The initial value of the *initial window size* GLUT state is 300 by 300. The *initial window size* components must be greater than zero.

The intent of the *initial window position* and *size* values is to provide a suggestion to the window system for a window's initial size and position. The window system is not obligated to use this information. Therefore, GLUT programs should not assume the window was created at the specified size or position. A GLUT program should use the window's reshape callback to determine the true size of the window.

2.3 glutInitDisplayMode

glutInitDisplayMode sets the *initial display mode*.

Usage

void glutInitDisplayMode(unsigned int mode);

mode Display mode, normally the bitwise OR-ing of GLUT display mode bit masks. See values below:

GLUT_RGBA Bit mask to select an RGBA mode window. This is the default if neither GLUT_RGBA nor GLUT_INDEX are specified.

GLUT_RGB An alias for GLUT_RGBA.

- GLUT_INDEX Bit mask to select a color index mode window. This overrides GLUT_RGBA if it is also specified.
- GLUT_SINGLE Bit mask to select a single buffered window. This is the default if neither GLUT_DOUBLE or GLUT_SINGLE are specified.
- GLUT_DOUBLE Bit mask to select a double buffered window. This overrides GLUT_SINGLE if it is also specified.

GLUT_ACCUM Bit mask to select a window with an accumulation buffer.

GLUT_ALPHA Bit mask to select a window with an alpha component to the color buffer(s).

GLUT_DEPTH Bit mask to select a window with a depth buffer.

GLUT_STENCIL Bit mask to select a window with a stencil buffer.

GLUT_MULTISAMPLE Bit mask to select a window with multisampling support. If multisampling is not available, a non-multisampling window will automatically be chosen. Note: both the OpenGL client-side and server-side implementations must support the GLX_SAMPLE_SGIS extension for multisampling to be available.

GLUT_STEREO Bit mask to select a stereo window.

GLUT_LUMINANCE Bit mask to select a window with a "luminance" color model. This model provides the functionality of OpenGL's RGBA color model, but the green and blue components are not maintained in the frame buffer. Instead each pixel's red component is converted to an index between zero and glutGet(GLUT_WINDOW_COLORMAP_SIZE) -1 and looked up in a per-window color map to determine the color of pixels within the window. The initial colormap of GLUT_LUMINANCE windows is initialized to be a linear gray ramp, but can be modified with GLUT's colormap routines.

Description

The *initial display mode* is used when creating top-level windows, subwindows, and overlays to determine the OpenGL display mode for the to-be-created window or overlay.

Note that GLUT_RGBA selects the RGBA color model, but it does not request any bits of alpha (sometimes called an *alpha buffer* or *destination alpha*) be allocated. To request alpha, specify GLUT_ALPHA. The same applies to GLUT_LUMINANCE.

GLUT_LUMINANCE Implementation Notes

GLUT_LUMINANCE is not supported on most OpenGL platforms.

3 Beginning Event Processing

After a GLUT program has done initial setup such as creating windows and menus, GLUT programs enter the GLUT event processing loop by calling glutMainLoop.

3.1 glutMainLoop

glutMainLoop enters the GLUT event processing loop.

Usage

```
void glutMainLoop(void);
```

Description

glutMainLoop enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

4 Window Management

GLUT supports two types of windows: top-level windows and subwindows. Both types support OpenGL rendering and GLUT callbacks. There is a single identifier space for both types of windows.

4.1 glutCreateWindow

4.1 glutCreateWindow

glutCreateWindow creates a top-level window.

Usage

int glutCreateWindow(char *name);

name ASCII character string for use as window name.

Description

glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name.

Implicitly, the current window is set to the newly created window.

Each created window has a unique associated OpenGL context. State changes to a window's associated OpenGL context can be done immediately after the window is created.

The *display state* of a window is initially for the window to be shown. But the window's *display state* is not actually acted upon until glutMainLoop is entered. This means until glutMainLoop is called, rendering to a created window is ineffective because the window can not yet be displayed.

The value returned is a unique small integer identifier for the window. The range of allocated identifiers starts at one. This window identifier can be used when calling glutSetWindow.

X Implementation Notes

The proper X Inter-Client Communication Conventions Manual (ICCCM) top-level properties are established. The WM_COMMAND property that lists the command line used to invoke the GLUT program is only established for the first window created.

4.2 glutCreateSubWindow

glutCreateSubWindow creates a subwindow.

Usage

win Identifier of the subwindow's parent window.

x Window X location in pixels relative to parent window's origin.

y Window Y location in pixels relative to parent window's origin.

width Width in pixels.

height Height in pixels.

Description

glutCreateSubWindow creates a subwindow of the window identified by win of size width and height at location x and y within the *current window*. Implicitly, the *current window* is set to the newly created subwindow.

Each created window has a unique associated OpenGL context. State changes to a window's associated OpenGL context can be done immediately after the window is created.

The *display state* of a window is initially for the window to be shown. But the window's *display state* is not actually acted upon until glutMainLoop is entered. This means until glutMainLoop is called, rendering to a created window is ineffective. Subwindows can not be iconified.

Subwindows can be nested arbitrarily deep.

The value returned is a unique small integer identifier for the window. The range of allocated identifiers starts at one.

4.3 glutSetWindow, glutGetWindow

glutSetWindow sets the *current window*; glutGetWindow returns the identifier of the *current window*.

Usage

```
void glutSetWindow(int win);
int glutGetWindow(void);
```

win Identifier of GLUT window to make the current window.

Description

glutSetWindow sets the *current window*; glutGetWindow returns the identifier of the *current window*. If no windows exist or the previously *current window* was destroyed, glutGetWindow returns zero. glutSetWindow does *not* change the *layer in use* for the window; this is done using glutUseLayer.

4.4 glutDestroyWindow

glutDestroyWindow destroys the specified window.

Usage

```
void glutDestroyWindow(int win);
```

win Identifier of GLUT window to destroy.

Description

glutDestroyWindow destroys the window specified by win and the window's associated OpenGL context, logical colormap (if the window is color index), and overlay and related state (if an overlay has been established). Any subwindows of destroyed windows are also destroyed by glutDestroyWindow. If win was the *current window*, the *current window* becomes invalid (glutGetWindow will return zero).

4.5 glutPostRedisplay

glutPostRedisplay marks the *current window* as needing to be redisplayed.

Usage

```
void glutPostRedisplay(void);
```

Description

Mark the normal plane of *current window* as needing to be redisplayed. The next iteration through glutMainLoop, the window's display callback will be called to redisplay the window's normal plane. Multiple calls to glutPostRedisplay before the next display callback opportunity generates only a single redisplay callback. glutPostRedisplay may be called within a window's display or overlay display callback to re-mark that window for redisplay.

Logically, normal plane damage notification for a window is treated as a glutPostRedisplay on the damaged window. Unlike damage reported by the window system, glutPostRedisplay will *not* set to true the normal plane's damaged status (returned by glutLayerGet(GLUT_NORMAL_DAMAGED).

Also, see glutPostOverlayRedisplay.

10

4.6 glutSwapBuffers

glutSwapBuffers swaps the buffers of the current window if double buffered.

Usage

```
void glutSwapBuffers(void);
```

Description

Performs a buffer swap on the *layer in use* for the *current window*. Specifically, glutSwapBuffers promotes the contents of the back buffer of the *layer in use* of the *current window* to become the contents of the front buffer. The contents of the back buffer then become undefined. The update typically takes place during the vertical retrace of the monitor, rather than immediately after glutSwapBuffers is called.

An implicit glFlush is done by glutSwapBuffers before it returns. Subsequent OpenGL commands can be issued immediately after calling glutSwapBuffers, but are not executed until the buffer exchange is completed.

If the *layer in use* is not double buffered, glutSwapBuffers has no effect.

4.7 glutPositionWindow

glutPositionWindow requests a change to the position of the current window.

Usage

```
void glutPositionWindow(int x, int y);
```

x New X location of window in pixels.

y New Y location of window in pixels.

Description

glutPositionWindow requests a change in the position of the *current window*. For top-level windows, the x and y parameters are pixel offsets from the screen origin. For subwindows, the x and y parameters are pixel offsets from the window origin.

The requests by glutPositionWindow are not processed immediately. The request is executed after returning to the main event loop. This allows multiple glutPositionWindow, glutReshapeWindow, and glutFullScreen requests to the same window to be coalesced.

In the case of top-level windows, a glutPositionWindow call is considered only a request for positioning the window. The window system is free to apply its own policies to top-level window placement. The intent is that top-level windows should be repositioned according glutPositionWindow's parameters.

glutPositionWindow disables the full screen status of a window if previously enabled.

4.8 glutReshapeWindow

glutReshapeWindow requests a change to the size of the *current window*.

Usage

```
void glutReshapeWindow(int width, int height);
```

width New width of window in pixels.

height New height of window in pixels.

Description

glutReshapeWindow requests a change in the size of the *current window*. The width and height parameters are size extents in pixels. The width and height must be positive values.

The requests by glutReshapeWindow are not processed immediately. The request is executed after returning to the main event loop. This allows multiple glutReshapeWindow, glutPositionWindow, and glutFullScreen requests to the same window to be coalesced.

In the case of top-level windows, a glutReshapeWindow call is considered only a request for sizing the window. The window system is free to apply its own policies to top-level window sizing. The intent is that top-level windows should be reshaped according glutReshapeWindow's parameters. Whether a reshape actually takes effect and, if so, the reshaped dimensions are reported to the program by a reshape callback.

glutReshapeWindow disables the full screen status of a window if previously enabled.

4.9 glutFullScreen

glutFullScreen requests that the *current window* be made full screen.

Usage

```
void glutFullScreen(void);
```

Description

glutFullScreen requests that the *current window* be made full screen. The exact semantics of what full screen means may vary by window system. The intent is to make the window as large as possible and disable any window decorations or borders added the window system. The window width and height are not guaranteed to be the same as the screen width and height, but that is the intent of making a window full screen.

glutFullScreen is defined to work only on top-level windows.

The glutFullScreen requests are not processed immediately. The request is executed after returning to the main event loop. This allows multiple glutReshapeWindow, glutPositionWindow, and glutFullScreen requests to the same window to be coalesced.

Subsequent glutReshapeWindow and glutPositionWindow requests on the window will disable the full screen status of the window.

X Implementation Notes

In the X implementation of GLUT, full screen is implemented by sizing and positioning the window to cover the entire screen and posting the _MOTIF_WM_HINTS property on the window requesting absolutely no decorations. Non-Motif window managers may not respond to _MOTIF_WM_HINTS.

4.10 glutPopWindow, glutPushWindow

glutPopWindow and glutPushWindow change the stacking order of the *current window* relative to its siblings.

Usage

```
void glutPopWindow(void);
void glutPushWindow(void);
```

Description

glutPopWindow and glutPushWindow work on both top-level windows and subwindows. The effect of pushing and popping windows does not take place immediately. Instead the push or pop is saved for execution upon return to the GLUT event loop. Subsequent push or pop requests on a window replace the previously

saved request for that window. The effect of pushing and popping top-level windows is subject to the window system's policy for restacking windows.

4.11 glutShowWindow, glutHideWindow, glutIconifyWindow

glutShowWindow, glutHideWindow, and glutIconifyWindow change the display status of the *cur*rent window.

Usage

```
void glutShowWindow(void);
void glutHideWindow(void);
void glutIconifyWindow(void);
```

Description

glutShowWindow will show the *current window* (though it may still not be visible if obscured by other shown windows). glutHideWindow will hide the *current window*. glutIconifyWindow will iconify a top-level window, but GLUT prohibits iconification of a subwindow. The effect of showing, hiding, and iconifying windows does not take place immediately. Instead the requests are saved for execution upon return to the GLUT event loop. Subsequent show, hide, or iconification requests on a window replace the previously saved request for that window. The effect of hiding, showing, or iconifying top-level windows is subject to the window system's policy for displaying windows.

4.12 glutSetWindowTitle, glutSetIconTitle

glutSetWindowTitle and glutSetIconTitle change the window or icon title respectively of the current top-level window.

Usage

```
void glutSetWindowTitle(char *name);
void glutSetIconTitle(char *name);
```

name ASCII character string for the window or icon name to be set for the window.

Description

These routines should be called only when the *current window* is a top-level window. Upon creation of a top-level window, the window and icon names are determined by the name parameter to glutCreateWindow. Once created, glutSetWindowTitle and glutSetIconTitle can change the window and icon names respectively of top-level windows. Each call requests the window system change the title appropriately. Requests are not buffered or coalesced. The policy by which the window and icon name are displayed is window system dependent.

4.13 glutSetCursor

glutSetCursor changes the cursor image of the current window.

Usage

void glutSetCursor(int cursor);

cursor Name of cursor image to change to.

GLUT_CURSOR_RIGHT_ARROW Arrow pointing up and to the right.

GLUT_CURSOR_LEFT_ARROW Arrow pointing up and to the left.

GLUT_CURSOR_INFO Pointing hand.

GLUT_CURSOR_DESTROY Skull & cross bones.

GLUT_CURSOR_HELP Question mark.

GLUT_CURSOR_CYCLE Arrows rotating in a circle.

GLUT_CURSOR_SPRAY Spray can.

GLUT_CURSOR_WAIT Wrist watch.

GLUT_CURSOR_TEXT Insertion point cursor for text.

GLUT_CURSOR_CROSSHAIR Simple cross-hair.

GLUT_CURSOR_UP_DOWN Bi-directional pointing up & down.

GLUT_CURSOR_LEFT_RIGHT Bi-directional pointing left & right.

GLUT_CURSOR_TOP_SIDE Arrow pointing to top side.

GLUT_CURSOR_BOTTOM_SIDE Arrow pointing to bottom side.

GLUT_CURSOR_LEFT_SIDE Arrow pointing to left side.

GLUT_CURSOR_RIGHT_SIDE Arrow pointing to right side.

GLUT_CURSOR_TOP_LEFT_CORNER Arrow pointing to top-left corner.

GLUT_CURSOR_TOP_RIGHT_CORNER Arrow pointing to top-right corner.

GLUT_CURSOR_BOTTOM_RIGHT_CORNER Arrow pointing to bottom-left corner.

GLUT_CURSOR_BOTTOM_LEFT_CORNER Arrow pointing to bottom-right corner.

GLUT_CURSOR_FULL_CROSSHAIR Full-screen cross-hair cursor (if possible, otherwise GLUT_CURSOR_CROSSHAIR).

GLUT_CURSOR_NONE Invisible cursor.

GLUT_CURSOR_INHERIT Use parent's cursor.

Description

glutSetCursor changes the cursor image of the *current window*. Each call requests the window system change the cursor appropriately. The cursor image when a window is created is GLUT_CURSOR_INHERIT. The exact cursor images used are implementation dependent. The intent is for the image to convey the meaning of the cursor name. For a top-level window, GLUT_CURSOR_INHERIT uses the default window system cursor.

X Implementation Notes

GLUT for X uses SGI's _SGI_CROSSHAIR_CURSOR convention [5] to access a full screen cross-hair cursor if possible.

5 Overlay Management

When overlay hardware is available, GLUT provides a set of routine for establishing, using, and removing an overlay for GLUT windows. When an overlay is established, a separate OpenGL context is also established. A window's overlay OpenGL state is kept distinct from the normal planes OpenGL state.

5.1 glutEstablishOverlay

glutEstablishOverlay establishes an overlay (if possible) for the *current window*.

5.2 glutUseLayer

Usage

void glutEstablishOverlay(void);

Description

glutEstablishOverlay establishes an overlay (if possible) for the current window. The requested display mode for the overlay is determined by the *initial display mode*. glutLayerGet(GLUT_OVERLAY_POSSIBLE) can be called to determine if an overlay is possible for the current window with the current initial display mode. Do not attempt to establish an overlay when one is not possible; GLUT will terminate the program.

If glutEstablishOverlay is called when an overlay already exists, the existing overlay is first removed, and then a new overlay is established. The state of the old overlay's OpenGL context is discarded.

The initial display state of an overlay is shown, however the overlay is only actually shown if the overlay's window is shown.

Implicitly, the window's layer in use changes to the overlay immediately after the overlay is established.

X Implementation Notes

GLUT for X uses the SERVER_OVERLAY_VISUALS convention [6] is used to determine if overlay visuals are available. While the convention allows for opaque overlays (no transparency) and overlays with the transparency specified as a bitmask, GLUT overlay management only provides access to transparent pixel overlays. Until RGBA overlays are better understood, GLUT only supports color index overlays.

5.2 glutUseLayer

glutUseLayer changes the *layer in use* for the *current window*.

Usage

void glutUseLayer(GLenum layer);

layer Either GLUT_NORMAL or GLUT_OVERLAY, selecting the normal plane or overlay respectively.

Description

glutUseLayer changes the per-window layer in use for the current window, selecting either the normal plane or overlay. The overlay should only be specified if an overlay exists, however windows without an overlay may still call glutUseLayer (GLUT_NORMAL). OpenGL commands for the window are directed to the current laver in use.

To query the *layer in use* for a window, call glutLayerGet(GLUT_LAYER_IN_USE).

5.3 glutRemoveOverlay

glutRemoveOverlay removes the overlay (if one exists) from the *current window*.

Usage

```
void glutRemoveOverlay(void);
```

Description

glutRemoveOverlay removes the overlay (if one exists). It is safe to call glutRemoveOverlay even if no overlay is currently established-it does nothing in this case. Implicitly, the window's layer in use changes to the normal plane immediately once the overlay is removed.

If the program intends to re-establish the overlay later, it is typically faster and less resource intensive to use glutHideOverlay and glutShowOverlay to simply change the display status of the overlay.

5.4 glutPostOverlayRedisplay

glutPostOverlayRedisplay marks the overlay of the current window as needing to be redisplayed.

Usage

```
void glutPostOverlayRedisplay(void);
```

Description

Mark the overlay of *current window* as needing to be redisplayed. The next iteration through glutMainLoop, the window's overlay display callback (or simply the display callback if no overlay display callback is registered) will be called to redisplay the window's overlay plane. Multiple calls to glutPostOverlayRedisplay before the next display callback opportunity (or overlay display callback opportunity if one is registered) generate only a single redisplay. glutPostOverlayRedisplay may be called within a window's display or overlay display callback to re-mark that window for redisplay.

Logically, overlay damage notification for a window is treated as a glutPostOverlayRedisplay on the damaged window. Unlike damage reported by the window system, glutPostOverlayRedisplay will not set to true the overlay's damaged status (returned by glutLayerGet(GLUT_OVERLAY_DAMAGED).

Also, see glutPostRedisplay.

5.5 glutShowOverlay, glutHideOverlay

glutShowOverlay shows the overlay of the current window; glutHideOverlay hides the overlay.

Usage

```
void glutShowOverlay(void);
void glutHideOverlay(void);
```

Description

glutShowOverlay shows the overlay of the *current window*; glutHideOverlay hides the overlay. The effect of showing or hiding an overlay takes place immediately. Note that glutShowOverlay will not actually display the overlay unless the window is also shown (and even a shown window may be obscured by other windows, thereby obscuring the overlay). It is typically faster and less resource intensive to use these routines to control the display status of an overlay as opposed to removing and re-establishing the overlay.

6 Menu Management

GLUT supports simple cascading pop-up menus. They are designed to let a user select various modes within a program. The functionality is simple and minimalistic and is meant to be that way. Do not mistake GLUT's pop-up menu facility with an attempt to create a full-featured user interface.

It is illegal to create or destroy menus, or change, add, or remove menu items while a menu (and any cascaded sub-menus) are in use (that is, popped up).

6.1 glutCreateMenu

glutCreateMenu creates a new pop-up menu.

Usage

```
int glutCreateMenu(void (*func)(int value));
```

func The callback function for the menu that is called when a menu entry from the menu is selected. The value passed to the callback is determined by the value for the selected menu entry.

6.2 glutSetMenu, glutGetMenu

Description

glutCreateMenu creates a new pop-up menu and returns a unique small integer identifier. The range of allocated identifiers starts at one. The menu identifier range is separate from the window identifier range. Implicitly, the *current menu* is set to the newly created menu. This menu identifier can be used when calling glutSetMenu.

When the menu callback is called because a menu entry is selected for the menu, the *current menu* will be implicitly set to the menu with the selected entry before the callback is made.

X Implementation Notes

If available, GLUT for X will take advantage of overlay planes for implementing pop-up menus. The use of overlay planes can eliminate display callbacks when pop-up menus are deactivated. The SERVER_OVERLAY_VISUALS convention [6] is used to determine if overlay visuals are available.

6.2 glutSetMenu, glutGetMenu

glutSetMenu sets the current menu; glutGetMenu returns the identifier of the current menu.

Usage

```
void glutSetMenu(int menu);
int glutGetMenu(void);
```

menu The identifier of the menu to make the current menu.

Description

glutSetMenu sets the *current menu*; glutGetMenu returns the identifier of the *current menu*. If no menus exist or the previous *current menu* was destroyed, glutGetMenu returns zero.

6.3 glutDestroyMenu

glutDestroyMenu destroys the specified menu.

Usage

void glutDestroyMenu(int menu);

menu The identifier of the menu to destroy.

Description

glutDestroyMenu destroys the specified menu by menu. If menu was the *current menu*, the *current menu* becomes invalid and glutGetMenu will return zero.

When a menu is destroyed, this has no effect on any sub-menus for which the destroyed menu has triggers. Sub-menu triggers are by name, not reference.

6.4 glutAddMenuEntry

glutAddMenuEntry adds a menu entry to the bottom of the *current menu*.

Usage

void glutAddMenuEntry(char *name, int value);

name ASCII character string to display in the menu entry.

value Value to return to the menu's callback function if the menu entry is selected.

Description

glutAddMenuEntry adds a menu entry to the bottom of the *current menu*. The string name will be displayed for the newly added menu entry. If the menu entry is selected by the user, the menu's callback will be called passing value as the callback's parameter.

6.5 glutAddSubMenu

glutAddSubMenu adds a sub-menu trigger to the bottom of the current menu.

Usage

void glutAddSubMenu(char *name, int menu);

name ASCII character string to display in the menu item from which to cascade the sub-menu.

menu Identifier of the menu to cascade from this sub-menu menu item.

Description

glutAddSubMenu adds a sub-menu trigger to the bottom of the *current menu*. The string name will be displayed for the newly added sub-menu trigger. If the sub-menu trigger is entered, the sub-menu numbered menu will be cascaded, allowing sub-menu menu items to be selected.

6.6 glutChangeToMenuEntry

glutChangeToMenuEntry changes the specified menu item in the current menu into a menu entry.

Usage

```
void glutChangeToMenuEntry(int entry, char *name, int value);
```

entry Index into the menu items of the current menu (1 is the topmost menu item).

name ASCII character string to display in the menu entry.

value Value to return to the menu's callback function if the menu entry is selected.

Description

glutChangeToMenuEntry changes the specified menu entry in the *current menu* into a menu entry. The entry parameter determines which menu item should be changed, with one being the topmost item. entry must be between 1 and glutGet(GLUT_MENU_NUM_ITEMS) inclusive. The menu item to change does not have to be a menu entry already. The string name will be displayed for the newly changed menu entry. The value will be returned to the menu's callback if this menu entry is selected.

6.7 glutChangeToSubMenu

glutChangeToSubMenu changes the specified menu item in the current menu into a sub-menu trigger.

Usage

```
void glutChangeToSubMenu(int entry, char *name, int menu);
```

entry Index into the menu items of the *current menu* (1 is the topmost menu item).

name ASCII character string to display in the menu item to cascade the sub-menu from.

menu Identifier of the menu to cascade from this sub-menu menu item.

6.8 glutRemoveMenuItem

Description

glutChangeToSubMenu changes the specified menu item in the *current menu* into a sub-menu trigger. The entry parameter determines which menu item should be changed, with one being the topmost item. entry must be between 1 and glutGet(GLUT_MENU_NUM_ITEMS) inclusive. The menu item to change does not have to be a sub-menu trigger already. The string name will be displayed for the newly changed sub-menu trigger. The menu identifier names the sub-menu to cascade from the newly added sub-menu trigger.

6.8 glutRemoveMenuItem

glutRemoveMenuItem remove the specified menu item.

Usage

```
void glutRemoveMenuItem(int entry);
```

entry Index into the menu items of the current menu (1 is the topmost menu item).

Description

glutRemoveMenuItem remove the entry menu item regardless of whether it is a menu entry or sub-menu trigger. entry must be between 1 and glutGet(GLUT_MENU_NUM_ITEMS) inclusive. Menu items below the removed menu item are renumbered.

6.9 glutAttachMenu, glutDetachMenu

glutAttachMenu attaches a mouse button for the *current window* to the identifier of the *current menu*; glutDetachMenu detaches an attached mouse button from the *current window*.

Usage

```
void glutAttachMenu(int button);
void glutDetachMenu(int button);
```

button The button to attach a menu or detach a menu.

Description

glutAttachMenu attaches a mouse button for the *current window* to the identifier of the *current menu*; glutDetachMenu detaches an attached mouse button from the *current window*. By attaching a menu identifier to a button, the named menu will be popped up when the user presses the specified button. button should be one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, and GLUT_RIGHT_BUTTON. Note that the menu is attached to the button by identifier, not by reference.

7 Callback Registration

GLUT supports a number of callbacks to respond to events. There are three types of callbacks: window, menu, and global. Window callbacks indicate when to redisplay or reshape a window, when the visibility of the window changes, and when input is available for the window. The menu callback is set by the glutCreateMenu call described already. The global callbacks manage the passing of time and menu usage. The calling order of callbacks between different windows is undefined.

Callbacks for input events should be delivered to the window the event occurs in. Events should not propagate to parent windows.

X Implementation Notes

The X GLUT implementation uses the X Input extension [13, 14] to support sophisticated input devices: Spaceball, dial & button box, and digitizing tablet. Because the X Input extension does not mandate how particular types of devices are advertised through the extension, it is possible GLUT for X may not correctly support input devices that would otherwise be of the correct type. The X GLUT implementation will support the Silicon Graphics Spaceball, dial & button box, and digitizing tablet as advertised through the X Input extension.

7.1 glutDisplayFunc

glutDisplayFunc sets the display callback for the *current window*.

Usage

```
void glutDisplayFunc(void (*func)(void));
```

func The new display callback function.

Description

glutDisplayFunc sets the display callback for the *current window*. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called. Before the callback, the *current window* is set to the window needing to be redisplayed and (if no overlay display callback is registered) the *layer in use* is set to the normal plane. The display callback is called with no parameters. The entire normal plane region should be redisplayed in response to the callback (this includes ancillary buffers if your program depends on their state).

GLUT determines when the display callback should be triggered based on the window's redisplay state. The redisplay state for a window can be either set explicitly by calling glutPostRedisplay or implicitly as the result of window damage reported by the window system. Multiple posted redisplays for a window are coalesced by GLUT to minimize the number of display callbacks called.

When an overlay is established for a window, but there is no overlay display callback registered, the display callback is used for redisplaying *both* the overlay and normal plane (that is, it will be called if either the redisplay state or overlay redisplay state is set). In this case, the *layer in use* is *not* implicitly changed on entry to the display callback.

See glutOverlayDisplayFunc to understand how distinct callbacks for the overlay and normal plane of a window may be established.

When a window is created, no display callback exists for the window. It is the responsibility of the programmer to install a display callback for the window before the window is shown. A display callback *must* be registered for any window that is shown. If a window becomes displayed without a display callback being registered, a fatal error occurs. Passing NULL to glutDisplayFunc is illegal as of GLUT 3.0; there is no way to "deregister" a display callback (though another callback routine can always be registered).

Upon return from the display callback, the *normal damaged* state of the window (returned by calling glutLayerGet(GLUT_NORMAL_DAMAGED) is cleared. If there is no overlay display callback registered the *overlay damaged* state of the window (returned by calling glutLayerGet(GLUT_OVERLAY_DAMAGED) is also cleared.

7.2 glutOverlayDisplayFunc

glutOverlayDisplayFunc sets the overlay display callback for the *current window*.

Usage

void glutOverlayDisplayFunc(void (*func)(void));

func The new overlay display callback function.

7.3 glutReshapeFunc

Description

glutDisplayFunc sets the overlay display callback for the *current window*. The overlay display callback is functionally the same as the window's display callback except that the overlay display callback is used to redisplay the window's overlay.

When GLUT determines that the overlay plane for the window needs to be redisplayed, the overlay display callback for the window is called. Before the callback, the *current window* is set to the window needing to be redisplayed and the *layer in use* is set to the overlay. The overlay display callback is called with no parameters. The entire overlay region should be redisplayed in response to the callback (this includes ancillary buffers if your program depends on their state).

GLUT determines when the overlay display callback should be triggered based on the window's overlay redisplay state. The overlay redisplay state for a window can be either set explicitly by calling glutPostOverlayRedisplay or implicitly as the result of window damage reported by the window system. Multiple posted overlay redisplays for a window are coalesced by GLUT to minimize the number of overlay display callbacks called.

Upon return from the overlay display callback, the *overlay damaged* state of the window (returned by calling glutLayerGet(GLUT_OVERLAY_DAMAGED) is cleared.

The overlay display callback can be deregistered by passing NULL to glutOverlayDisplayFunc. The overlay display callback is initially NULL when an overlay is established. See glutDisplayFunc to understand how the display callback alone is used if an overlay display callback is not registered.

7.3 glutReshapeFunc

glutReshapeFunc sets the reshape callback for the *current window*.

Usage

void glutReshapeFunc(void (*func)(int width, int height));

func The new reshape callback function.

Description

glutReshapeFunc sets the reshape callback for the *current window*. The reshape callback is triggered when a window is reshaped. A reshape callback is also triggered immediately before a window's first display callback after a window is created or whenever an overlay for the window is established. The width and height parameters of the callback specify the new window size in pixels. Before the callback, the *current window* is set to the window that has been reshaped.

If a reshape callback is not registered for a window or NULL is passed to glutReshapeFunc (to deregister a previously registered callback), the default reshape callback is used. This default callback will simply call glViewport(0,0,width,height) on the normal plane (and on the overlay if one exists).

If an overlay is established for the window, a single reshape callback is generated. It is the callback's responsibility to update both the normal plane and overlay for the window (changing the *layer in use* as necessary).

When a top-level window is reshaped, subwindows are not reshaped. It is up to the GLUT program to manage the size and positions of subwindows within a top-level window. Still, reshape callbacks will be triggered for subwindows when their size is changed using glutReshapeWindow.

7.4 glutKeyboardFunc

glutKeyboardFunc sets the keyboard callback for the *current window*.

Usage

```
void glutKeyboardFunc(void (*func)(unsigned char key,
```

```
int x, int y));
```

func The new keyboard callback function.

Description

glutKeyboardFunc sets the keyboard callback for the *current window*. When a user types into the window, each key press generating an ASCII character will generate a keyboard callback. The key callback parameter is the generated ASCII character. The state of modifier keys such as Shift cannot be determined directly; their only effect will be on the returned ASCII data. The x and y callback parameters indicate the mouse location in window relative coordinates when the key was pressed. When a new window is created, no keyboard callback is initially registered, and ASCII key strokes in the window are ignored. Passing NULL to glutKeyboardFunc disables the generation of keyboard callbacks.

During a keyboard callback, glutGetModifiers may be called to determine the state of modifier keys when the keystroke generating the callback occurred.

Also, see glutSpecialFunc for a means to detect non-ASCII key strokes.

7.5 glutMouseFunc

glutMouseFunc sets the mouse callback for the *current window*.

Usage

func The new mouse callback function.

Description

glutMouseFunc sets the mouse callback for the *current window*. When a user presses and releases mouse buttons in the window, each press and each release generates a mouse callback. The button parameter is one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. For systems with only two mouse buttons, it may not be possible to generate GLUT_MIDDLE_BUTTON callback. For systems with a single mouse button, it may be possible to generate only a GLUT_LEFT_BUTTON callback. The state parameter is either GLUT_UP or GLUT_DOWN indicating whether the callback was due to a release or press respectively. The x and y callback parameters indicate the window relative coordinates when the mouse button state changed. If a GLUT_DOWN callback for a specific button is triggered, the program can assume a GLUT_UP callback for the same button will be generated (assuming the window still has a mouse callback registered) when the mouse button is released even if the mouse has moved outside the window.

If a menu is attached to a button for a window, mouse callbacks will not be generated for that button.

During a mouse callback, glutGetModifiers may be called to determine the state of modifier keys when the mouse event generating the callback occurred.

Passing NULL to glutMouseFunc disables the generation of mouse callbacks.

7.6 glutMotionFunc, glutPassiveMotionFunc

glutMotionFunc and glutPassiveMotionFunc set the motion and passive motion callbacks respectively for the *current window*.

Usage

```
void glutMotionFunc(void (*func)(int x, int y));
void glutPassiveMotionFunc(void (*func)(int x, int y));
```

func The new motion or passive motion callback function.

7.7 glutVisibilityFunc

Description

glutMotionFunc and glutPassiveMotionFunc set the motion and passive motion callback respectively for the *current window*. The motion callback for a window is called when the mouse moves within the window while one or more mouse buttons are pressed. The passive motion callback for a window is called when the mouse moves within the window while *no* mouse buttons are pressed.

The x and y callback parameters indicate the mouse location in window relative coordinates.

Passing NULL to glutMotionFunc or glutPassiveMotionFunc disables the generation of the mouse or passive motion callback respectively.

7.7 glutVisibilityFunc

glutVisibilityFunc sets the visibility callback for the *current window*.

Usage

```
void glutVisibilityFunc(void (*func)(int state));
```

func The new visibility callback function.

Description

glutVisibilityFunc sets the visibility callback for the *current window*. The visibility callback for a window is called when the visibility of a window changes. The state callback parameter is either GLUT_NOT_VISIBLE or GLUT_VISIBLE depending on the current visibility of the window. GLUT_VISIBLE does not distinguish a window being totally versus partially visible. GLUT_NOT_VISIBLE means no part of the window is visible, i.e., until the window's visibility changes, all further rendering to the window is discarded.

GLUT considers a window visible if any pixel of the window is visible *or* any pixel of any descendant window is visible on the screen.

Passing NULL to glutVisibilityFunc disables the generation of the visibility callback.

If the visibility callback for a window is disabled and later re-enabled, the visibility status of the window is undefined; any change in window visibility will be reported, that is if you disable a visibility callback and re-enable the callback, you are guaranteed the next visibility change will be reported.

7.8 glutEntryFunc

glutEntryFunc sets the mouse enter/leave callback for the *current window*.

Usage

```
void glutEntryFunc(void (*func)(int state));
```

func The new entry callback function.

Description

glutEntryFunc sets the mouse enter/leave callback for the *current window*. The state callback parameter is either GLUT_LEFT or GLUT_ENTERED depending on if the mouse pointer has last left or entered the window.

Passing NULL to glutEntryFunc disables the generation of the mouse enter/leave callback. Some window systems may not generate accurate enter/leave callbacks.

X Implementation Notes

An X implementation of GLUT should generate accurate enter/leave callbacks.

7.9 glutSpecialFunc

glutSpecialFunc sets the special keyboard callback for the *current window*.

Usage

```
void glutSpecialFunc(void (*func)(int key, int x, int y));
```

func The new special callback function.

Description

glutSpecialFunc sets the special keyboard callback for the *current window*. The special keyboard callback is triggered when keyboard function or directional keys are pressed. The key callback parameter is a GLUT_KEY_* constant for the special key pressed. The x and y callback parameters indicate the mouse in window relative coordinates when the key was pressed. When a new window is created, no special callback is initially registered and special key strokes in the window are ignored. Passing NULL to glutSpecialFunc disables the generation of special callbacks.

During a special callback, glutGetModifiers may be called to determine the state of modifier keys when the keystroke generating the callback occurred.

An implementation should do its best to provide ways to generate all the GLUT_KEY_* special keys. The available GLUT_KEY_* values are:

GLUT_KEY_F1 F1 function key.

- GLUT_KEY_F2 F2 function key.
- GLUT_KEY_F3 F3 function key.
- GLUT_KEY_F4 F4 function key.
- GLUT_KEY_F5 F5 function key.
- GLUT_KEY_F6 F6 function key.
- GLUT_KEY_F7 F7 function key.
- GLUT_KEY_F8 F8 function key.
- GLUT_KEY_F9 F9 function key.
- GLUT_KEY_F10 F10 function key.
- GLUT_KEY_F11 F11 function key.

GLUT_KEY_F12 F12 function key.

- GLUT_KEY_LEFT Left directional key.
- GLUT_KEY_UP Up directional key.

GLUT_KEY_RIGHT Right directional key.

GLUT_KEY_DOWN Down directional key.

GLUT_KEY_PAGE_UP Page up directional key.

GLUT_KEY_PAGE_DOWN Page down directional key.

GLUT_KEY_HOME Home directional key.

GLUT_KEY_END End directional key.

GLUT_KEY_INSERT Inset directional key.

Note that the escape, backspace, and delete keys are generated as an ASCII character.

7.10 glutSpaceballMotionFunc

glutSpaceballMotionFunc sets the Spaceball motion callback for the *current window*.

7.11 glutSpaceballRotateFunc

Usage

```
void glutSpaceballMotionFunc(void (*func)(int x, int y, int z));
```

func The new spaceball motion callback function.

Description

glutSpaceballMotionFunc sets the Spaceball motion callback for the *current window*. The Spaceball motion callback for a window is called when the window has Spaceball input focus (normally, when the mouse is in the window) and the user generates Spaceball translations. The x, y, and z callback parameters indicate the translations along the X, Y, and Z axes. The callback parameters are normalized to be within the range of -1000 to 1000 inclusive.

Registering a Spaceball motion callback when a Spaceball device is not available has no effect and is not an error. In this case, no Spaceball motion callbacks will be generated.

Passing NULL to glutSpaceballMotionFunc disables the generation of Spaceball motion callbacks. When a new window is created, no Spaceball motion callback is initially registered.

7.11 glutSpaceballRotateFunc

glutSpaceballRotateFunc sets the Spaceball rotation callback for the *current window*.

Usage

```
void glutSpaceballRotateFunc(void (*func)(int x, int y, int z));
```

func The new spaceball rotate callback function.

Description

glutSpaceballRotateFunc sets the Spaceball rotate callback for the *current window*. The Spaceball rotate callback for a window is called when the window has Spaceball input focus (normally, when the mouse is in the window) and the user generates Spaceball rotations. The x, y, and z callback parameters indicate the rotation along the X, Y, and Z axes. The callback parameters are normalized to be within the range of -1800 to 1800 inclusive.

Registering a Spaceball rotate callback when a Spaceball device is not available is ineffectual and not an error. In this case, no Spaceball rotate callbacks will be generated.

Passing NULL to glutSpaceballRotateFunc disables the generation of Spaceball rotate callbacks. When a new window is created, no Spaceball rotate callback is initially registered.

7.12 glutSpaceballButtonFunc

glutSpaceballButtonFunc sets the Spaceball button callback for the *current window*.

Usage

```
void glutSpaceballButtonFunc(void (*func)(int button, int state));
```

func The new spaceball button callback function.

Description

glutSpaceballButtonFunc sets the Spaceball button callback for the *current window*. The Spaceball button callback for a window is called when the window has Spaceball input focus (normally, when the mouse is in the window) and the user generates Spaceball button presses. The button parameter will be the button number (starting at one). The number of available Spaceball buttons can be determined with glutDeviceGet (GLUT_NUM_SPACEBALL_BUTTONS). The state is either GLUT_UP or GLUT_DOWN indicating whether the callback was due to a release or press respectively.

Registering a Spaceball button callback when a Spaceball device is not available is ineffectual and not an error. In this case, no Spaceball button callbacks will be generated.

Passing NULL to glutSpaceballButtonFunc disables the generation of Spaceball button callbacks. When a new window is created, no Spaceball button callback is initially registered.

7.13 glutButtonBoxFunc

glutButtonBoxFunc sets the dial & button box button callback for the *current window*.

Usage

```
void glutButtonBoxFunc(void (*func)(int button, int state));
```

func The new button box callback function.

Description

glutButtonBoxFunc sets the dial & button box button callback for the *current window*. The dial & button box button callback for a window is called when the window has dial & button box input focus (normally, when the mouse is in the window) and the user generates dial & button box button presses. The button parameter will be the button number (starting at one). The number of available dial & button box buttons can be determined with glutDeviceGet(GLUT_NUM_BUTTON_BOX_BUTTONS). The state is either GLUT_UP or GLUT_DOWN indicating whether the callback was due to a release or press respectively.

Registering a dial & button box button callback when a dial & button box device is not available is ineffectual and not an error. In this case, no dial & button box button callbacks will be generated.

Passing NULL to glutButtonBoxFunc disables the generation of dial & button box button callbacks. When a new window is created, no dial & button box button callback is initially registered.

7.14 glutDialsFunc

glutDialsFunc sets the dial & button box dials callback for the *current window*.

Usage

```
void glutDialsFunc(void (*func)(int dial, int value));
```

func The new dials callback function.

Description

glutDialsFunc sets the dial & button box dials callback for the *current window*. The dial & button box dials callback for a window is called when the window has dial & button box input focus (normally, when the mouse is in the window) and the user generates dial & button box dial changes. The dial parameter will be the dial number (starting at one). The number of available dial & button box dials can be determined with glutDeviceGet(GLUT_NUM_DIALS). The value measures the absolute rotation in degrees. Dial values do not "roll over" with each complete rotation but continue to accumulate degrees (until the int dial value overflows).

Registering a dial & button box dials callback when a dial & button box device is not available is ineffectual and not an error. In this case, no dial & button box dials callbacks will be generated.

Passing NULL to glutDialsFunc disables the generation of dial & button box dials callbacks. When a new window is created, no dial & button box dials callback is initially registered.

26

7.15 glutTabletMotionFunc

7.15 glutTabletMotionFunc

glutTabletMotionFunc sets the special keyboard callback for the *current window*.

Usage

```
void glutTabletMotionFunc(void (*func)(int x, int y));
```

func The new tablet motion callback function.

Description

glutTabletMotionFunc sets the tablet motion callback for the *current window*. The tablet motion callback for a window is called when the window has tablet input focus (normally, when the mouse is in the window) and the user generates tablet motion. The x and y callback parameters indicate the absolute position of the tablet "puck" on the tablet. The callback parameters are normalized to be within the range of 0 to 2000 inclusive.

Registering a tablet motion callback when a tablet device is not available is ineffectual and not an error. In this case, no tablet motion callbacks will be generated.

Passing NULL to glutTabletMotionFunc disables the generation of tablet motion callbacks. When a new window is created, no tablet motion callback is initially registered.

7.16 glutTabletButtonFunc

glutTabletButtonFunc sets the special keyboard callback for the *current window*.

Usage

func The new tablet button callback function.

Description

glutTabletButtonFunc sets the tablet button callback for the *current window*. The tablet button callback for a window is called when the window has tablet input focus (normally, when the mouse is in the window) and the user generates tablet button presses. The button parameter will be the button number (starting at one). The number of available tablet buttons can be determined with glutDeviceGet(GLUT_NUM_TABLET_BUTTONS). The state is either GLUT_UP or GLUT_DOWN indicating whether the callback was due to a release or press respectively. The x and y callback parameters indicate the window relative coordinates when the tablet button state changed.

Registering a tablet button callback when a tablet device is not available is ineffectual and not an error. In this case, no tablet button callbacks will be generated.

Passing NULL to glutTabletButtonFunc disables the generation of tablet button callbacks. When a new window is created, no tablet button callback is initially registered.

7.17 glutMenuStatusFunc

glutMenuStatusFunc sets the global menu status callback.

Usage

```
void glutMenuStatusFunc(void (*func)(int status, int x, int y));
void glutMenuStateFunc(void (*func)(int status));
```

func The new menu status (or state) callback function.

Description

glutMenuStatusFunc sets the global menu status callback so a GLUT program can determine when a menu is in use or not. When a menu status callback is registered, it will be called with the value GLUT_MENU_IN_USE for its value parameter when pop-up menus are in use by the user; and the callback will be called with the value GLUT_MENU_NOT_IN_USE for its status parameter when pop-up menus are no longer in use. The x and y parameters indicate the location in window coordinates of the button press that caused the menu to go into use, or the location where the menu was released (may be outside the window). The func parameter names the callback function. Other callbacks continue to operate (except mouse motion callbacks) when pop-up menus are in use so the menu status callback allows a program to suspend animation or other tasks when menus are in use. The cascading and unmapping of sub-menus from an initial pop-up menu does not generate menu status callbacks for GLUT.

When the menu status callback is called, the *current menu* will be set to the initial pop-up menu in both the GLUT_MENU_IN_USE and GLUT_MENU_NOT_IN_USE cases. The *current window* will be set to the window from which the initial menu was popped up from, also in both cases.

Passing NULL to glutMenuStatusFunc disables the generation of the menu status callback.

glutMenuStateFunc is a deprecated version of the glutMenuStatusFunc routine. The only difference is glutMenuStateFunc callback prototype does not deliver the two additional x and y coordinates.

7.18 glutIdleFunc

glutIdleFunc sets the global idle callback.

Usage

void glutIdleFunc(void (*func)(void));

func The new idle callback function.

Description

glutIdleFunc sets the global idle callback to be func so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received. If enabled, the idle callback is continuously called when events are not being received. The callback routine has no parameters. The *current window* and *current menu* will not be changed before the idle callback. Programs with multiple windows and/or menus should explicitly set the *current window* and/or *current menu* and not rely on its current setting.

The amount of computation and rendering done in an idle callback should be minimized to avoid affecting the program's interactive response. In general, not more than a single frame of rendering should be done in an idle callback.

Passing NULL to glutIdleFunc disables the generation of the idle callback.

7.19 glutTimerFunc

glutTimerFunc registers a timer callback to be triggered in a specified number of milliseconds.

Usage

msecs Number of milliseconds to pass before calling the callback.

func The timer callback function.

value Integer value to pass to the timer callback.

Description

glutTimerFunc registers the timer callback func to be triggered in at least msecs milliseconds. The value parameter to the timer callback will be the value of the value parameter to glutTimerFunc. Multiple timer callbacks at same or differing times may be registered simultaneously.

The number of milliseconds is a lower bound on the time before the callback is generated. GLUT attempts to deliver the timer callback as soon as possible after the expiration of the callback's time interval.

There is no support for canceling a registered callback. Instead, ignore a callback based on its value parameter when it is triggered.

8 Color Index Colormap Management

OpenGL supports both RGBA and color index rendering. The RGBA mode is generally preferable to color index because more OpenGL rendering capabilities are available and color index mode requires the loading of colormap entries.

The GLUT color index routines are used to write and read entries in a window's color index colormap. Every GLUT color index window has its own logical color index colormap. The size of a window's colormap can be determined by calling glutGet(GLUT_WINDOW_COLORMAP_SIZE).

GLUT color index windows within a program can attempt to share colormap resources by copying a single color index colormap to multiple windows using glutCopyColormap. If possible GLUT will attempt to share the actual colormap. While copying colormaps using glutCopyColormap can potentially allow sharing of physical colormap resources, logically each window has its own colormap. So changing a copied colormap of a window will force the duplication of the colormap. For this reason, color index programs should generally load a single color index colormap, copy it to all color index windows within the program, and then not modify any colormap cells.

Use of multiple colormaps is likely to result in colormap installation problems where some windows are displayed with an incorrect colormap due to limitations on colormap resources.

8.1 glutSetColor

glutSetColor sets the color of a colormap entry in the layer of use for the current window.

Usage

cell Color cell index (starting at zero).

red Red intensity (clamped between 0.0 and 1.0 inclusive).

green Green intensity (clamped between 0.0 and 1.0 inclusive).

blue Blue intensity (clamped between 0.0 and 1.0 inclusive).

Description

Sets the cell color index colormap entry of the *current window*'s logical colormap for the *layer in use* with the color specified by red, green, and blue. The *layer in use* of the *current window* should be a color index window. cell should be zero or greater and less than the total number of colormap entries for the window. If the *layer in use*'s colormap was copied by reference, a glutSetColor call will force the duplication of the colormap. Do not attempt to set the color of an overlay's transparent index.

8.2 glutGetColor

glutGetColor retrieves a red, green, or blue component for a given color index colormap entry for the *layer in use*'s logical colormap for the *current window*.

Usage

GLfloat glutGetColor(int cell, int component);

cell Color cell index (starting at zero).

component One of GLUT_RED, GLUT_GREEN, or GLUT_BLUE.

Description

glutGetColor retrieves a red, green, or blue component for a given color index colormap entry for the *current window*'s logical colormap. The *current window* should be a color index window. cell should be zero or greater and less than the total number of colormap entries for the window. For valid color indices, the value returned is a floating point value between 0.0 and 1.0 inclusive. glutGetColor will return -1.0 if the color index specified is an overlay's transparent index, less than zero, or greater or equal to the value returned by glutGet(GLUT_WINDOW_COLORMAP_SIZE), that is if the color index is transparent or outside the valid range of color indices.

8.3 glutCopyColormap

glutCopyColormap copies the logical colormap for the *layer in use* from a specified window to the *current window*.

Usage

void glutCopyColormap(int win);

win The identifier of the window to copy the logical colormap from.

Description

glutCopyColormap copies (lazily if possible to promote sharing) the logical colormap from a specified window to the *current window's layer in use*. The copy will be from the normal plane to the normal plane; or from the overlay to the overlay (never across different layers). Once a colormap has been copied, avoid setting cells in the colormap with glutSetColor since that will force an actual copy of the colormap if it was previously copied by reference. glutCopyColormap should only be called when both the *current window* and the win window are color index windows.

9 State Retrieval

GLUT maintains a considerable amount of programmer visible state. Some (but not all) of this state may be directly retrieved.

9.1 glutGet

glutGet retrieves simple GLUT state represented by integers.

Usage

int glutGet(GLenum state);

state Name of state to retrieve.

GLUT_WINDOW_X X location in pixels (relative to the screen origin) of the current window.

GLUT_WINDOW_Y Y location in pixels (relative to the screen origin) of the *current window*. GLUT_WINDOW_WIDTH Width in pixels of the *current window*.

30

GLUT_WINDOW_HEIGHT Height in pixels of the current window.

- GLUT_WINDOW_BUFFER_SIZE Total number of bits for *current window*'s color buffer. For an RGBA window, this is the sum of GLUT_WINDOW_RED_SIZE, GLUT_WINDOW_GREEN_SIZE, GLUT_WINDOW_BLUE_SIZE, and GLUT_WINDOW_ALPHA_SIZE. For color index windows, this is the number of bits for color indices.
- GLUT_WINDOW_STENCIL_SIZE Number of bits in the *current window*'s stencil buffer.
- GLUT_WINDOW_DEPTH_SIZE Number of bits in the *current window*'s depth buffer.
- GLUT_WINDOW_RED_SIZE Number of bits of red stored the *current window*'s color buffer. Zero if the window is color index.
- GLUT_WINDOW_GREEN_SIZE Number of bits of green stored the *current window*'s color buffer. Zero if the window is color index.
- GLUT_WINDOW_BLUE_SIZE Number of bits of blue stored the *current window*'s color buffer. Zero if the window is color index.
- GLUT_WINDOW_ALPHA_SIZE Number of bits of alpha stored the *current window*'s color buffer. Zero if the window is color index.
- GLUT_WINDOW_ACCUM_RED_SIZE Number of bits of red stored in the *current window*'s accumulation buffer. Zero if the window is color index.
- GLUT_WINDOW_ACCUM_GREEN_SIZE Number of bits of green stored in the *current window*'s accumulation buffer. Zero if the window is color index.
- GLUT_WINDOW_ACCUM_BLUE_SIZE Number of bits of blue stored in the *current window*'s accumulation buffer. Zero if the window is color index.
- GLUT_WINDOW_ACCUM_ALPHA_SIZE Number of bits of alpha stored in the *current window*'s accumulation buffer. Zero if the window is color index.
- GLUT_WINDOW_DOUBLEBUFFER One if the current window is double buffered, zero otherwise.
- GLUT_WINDOW_RGBA One if the current window is RGBA mode, zero otherwise (i.e., color index).
- GLUT_WINDOW_PARENT The window number of the *current window*'s parent; zero if the window is a top-level window.
- GLUT_WINDOW_NUM_CHILDREN The number of subwindows the *current window* has (not counting children of children).
- GLUT_WINDOW_COLORMAP_SIZE Size of *current window*'s color index colormap; zero for RGBA color model windows.
- GLUT_WINDOW_NUM_SAMPLES Number of samples for multisampling for the current window.
- GLUT_WINDOW_STEREO One if the *current window* is stereo, zero otherwise.
- GLUT_WINDOW_CURSOR Current cursor for the *current window*.
- GLUT_SCREEN_WIDTH Width of the screen in pixels. Zero indicates the width is unknown or not available.
- GLUT_SCREEN_HEIGHT Height of the screen in pixels. Zero indicates the height is unknown or not available.
- GLUT_SCREEN_WIDTH_MM Width of the screen in millimeters. Zero indicates the width is unknown or not available.
- GLUT_SCREEN_HEIGHT_MM Height of the screen in millimeters. Zero indicates the height is unknown or not available.
- GLUT_MENU_NUM_ITEMS Number of menu items in the *current menu*.
- GLUT_DISPLAY_MODE_POSSIBLE Whether the *current display mode* is supported or not.
- GLUT_INIT_DISPLAY_MODE The *initial display mode* bit mask.
- GLUT_INIT_WINDOW_X The X value of the *initial window position*.
- GLUT_INIT_WINDOW_Y The Y value of the *initial window position*.

GLUT_INIT_WINDOW_WIDTH The width value of the *initial window size*.

GLUT_INIT_WINDOW_HEIGHT The height value of the *initial window size*.

GLUT_ELAPSED_TIME Number of milliseconds since glutInit called (or first call to glutGet(GLUT_ELAPSED_TIME)).

Description

glutGet retrieves simple GLUT state represented by integers. The state parameter determines what type of state to return. Window capability state is returned for the *layer in use*. GLUT state names beginning with GLUT_WINDOW_ return state for the *current window*. GLUT state names beginning with GLUT_MENU_ return state for the *current menu*. Other GLUT state names return global state. Requesting state for an invalid GLUT state name returns negative one.

9.2 glutLayerGet

glutLayerGet retrieves GLUT state pertaining to the layers of the current window.

Usage

int glutLayerGet(GLenum info);

info Name of device information to retrieve.

- GLUT_OVERLAY_POSSIBLE Whether an overlay could be established for the *current window* given the current *initial display mode*. If false, glutEstablishOverlay will fail with a fatal error if called.
- GLUT_LAYER_IN_USE Either GLUT_NORMAL or GLUT_OVERLAY depending on whether the normal plane or overlay is the *layer in use*.
- GLUT_HAS_OVERLAY If the current window has an overlay established.
- GLUT_TRANSPARENT_INDEX The transparent color index of the overlay of the *current window*; negative one is returned if no overlay is in use.
- GLUT_NORMAL_DAMAGED True if the normal plane of the *current window* has damaged (by window system activity) since the last display callback was triggered. Calling glutPostRedisplay will not set this true.
- GLUT_OVERLAY_DAMAGED True if the overlay plane of the *current window* has damaged (by window system activity) since the last display callback was triggered. Calling glutPostRedisplay or glutPostOverlayRedisplay will not set this true. Negative one is returned if no overlay is in use.

Description

glutLayerGet retrieves GLUT layer information for the *current window* represented by integers. The info parameter determines what type of layer information to return.

9.3 glutDeviceGet

glutDeviceGet retrieves GLUT device information represented by integers.

Usage

int glutDeviceGet(GLenum info);

info Name of device information to retrieve.

GLUT_HAS_KEYBOARD Non-zero if a keyboard is available; zero if not available. For most GLUT implementations, a keyboard can be assumed.

- GLUT_HAS_MOUSE Non-zero if a mouse is available; zero if not available. For most GLUT implementations, a keyboard can be assumed.
- GLUT_HAS_SPACEBALL Non-zero if a Spaceball is available; zero if not available.

GLUT_HAS_DIAL_AND_BUTTON_BOX Non-zero if a dial & button box is available; zero if not available.

- GLUT_HAS_TABLET Non-zero if a tablet is available; zero if not available.
- GLUT_NUM_MOUSE_BUTTONS Number of buttons supported by the mouse. If no mouse is supported, zero is returned.
- GLUT_NUM_SPACEBALL_BUTTONS Number of buttons supported by the Spaceball. If no Spaceball is supported, zero is returned.
- GLUT_NUM_BUTTON_BOX_BUTTONS Number of buttons supported by the dial & button box device. If no dials & button box device is supported, zero is returned.
- GLUT_NUM_DIALS Number of dials supported by the dial & button box device. If no dials & button box device is supported, zero is returned.
- GLUT_NUM_TABLET_BUTTONS Number of buttons supported by the tablet. If no tablet is supported, zero is returned.

Description

glutDeviceGet retrieves GLUT device information represented by integers. The info parameter determines what type of device information to return. Requesting device information for an invalid GLUT device information name returns negative one.

9.4 glutGetModifiers

glutGetModifiers returns the modifier key state when certain callbacks were generated.

Usage

```
int glutGetModifiers(void);
```

GLUT_ACTIVE_SHIFT Set if the Shift modifier or Caps Lock is active.

GLUT_ACTIVE_CTRL Set if the Ctrl modifier is active.

GLUT_ACTIVE_ALT Set if the Alt modifier is active.

Description

glutGetModifiers returns the modifier key state at the time the input event for a keyboard, special, or mouse callback is generated. This routine may only be called while a keyboard, special, or mouse callback is being handled. The window system is permitted to intercept window system defined modifier key strokes or mouse buttons, in which case, no GLUT callback will be generated. This interception will be independent of use of glutGetModifiers.

9.5 glutExtensionSupported

glutExtensionSupported helps to easily determine whether a given OpenGL extension is supported.

Usage

int glutExtensionSupported(char *extension);

extension Name of OpenGL extension.

Description

glutExtensionSupported helps to easily determine whether a given OpenGL extension is supported or not. The extension parameter names the extension to query. The supported extensions can also be determined with glGetString(GL_EXTENSIONS), but glutExtensionSupported does the correct parsing of the returned string.

glutExtensionSupported returns non-zero if the extension is supported, zero if not supported.

There must be a valid *current window* to call glutExtensionSupported.

glutExtensionSupported only returns information about OpenGL extensions only. This means window system dependent extensions (for example, GLX extensions) are not reported by glutExtensionSupported.

10 Font Rendering

GLUT supports two type of font rendering: stroke fonts, meaning each character is rendered as a set of line segments; and bitmap fonts, where each character is a bitmap generated with glBitmap. Stroke fonts have the advantage that because they are geometry, they can be arbitrarily scale and rendered. Bitmap fonts are less flexible since they are rendered as bitmaps but are usually faster than stroke fonts.

10.1 glutBitmapCharacter

glutBitmapCharacter renders a bitmap character using OpenGL.

Usage

```
void glutBitmapCharacter(void *font, int character);
```

font Bitmap font to use.

character Character to render (not confined to 8 bits).

Description

Without using any display lists, glutBitmapCharacter renders the character in the named bitmap font. The available fonts are:

GLUT_BITMAP_8_BY_13 A fixed width font with every character fitting in an 8 by 13 pixel rectangle. The exact bitmaps to be used is defined by the standard X glyph bitmaps for the X font named:

-misc-fixed-medium-r-normal--13-120-75-75-C-80-iso8859-1

GLUT_BITMAP_9_BY_15 A fixed width font with every character fitting in an 9 by 15 pixel rectangle. The exact bitmaps to be used is defined by the standard X glyph bitmaps for the X font named:

-misc-fixed-medium-r-normal--15-140-75-75-C-90-iso8859-1

GLUT_BITMAP_TIMES_ROMAN_10 A 10-point proportional spaced Times Roman font. The exact bitmaps to be used is defined by the standard X glyph bitmaps for the X font named:

-adobe-times-medium-r-normal--10-100-75-75-p-54-iso8859-1

GLUT_BITMAP_TIMES_ROMAN_24 A 24-point proportional spaced Times Roman font. The exact bitmaps to be used is defined by the standard X glyph bitmaps for the X font named:

-adobe-times-medium-r-normal--24-240-75-75-p-124-iso8859-1

GLUT_BITMAP_HELVETICA_10 A 10-point proportional spaced Helvetica font. The exact bitmaps to be used is defined by the standard X glyph bitmaps for the X font named:

-adobe-helvetica-medium-r-normal--10-100-75-75-p-56-iso8859-1

GLUT_BITMAP_HELVETICA_12 A 12-point proportional spaced Helvetica font. The exact bitmaps to be used is defined by the standard X glyph bitmaps for the X font named:

-adobe-helvetica-medium-r-normal--12-120-75-75-p-67-iso8859-1

GLUT_BITMAP_HELVETICA_18 A 18-point proportional spaced Helvetica font. The exact bitmaps to be used is defined by the standard X glyph bitmaps for the X font named:

-adobe-helvetica-medium-r-normal--18-180-75-75-p-98-iso8859-1

Rendering a nonexistent character has no effect. glutBitmapCharacter automatically sets the OpenGL unpack pixel storage modes it needs appropriately and saves and restores the previous modes before returning. The generated call to glBitmap will adjust the current raster position based on the width of the character.

10.2 glutBitmapWidth

glutBitmapWidth returns the width of a bitmap character.

Usage

```
int glutBitmapWidth(GLUTbitmapFont font, int character);
```

font Bitmap font to use.

character Character to return width of (not confined to 8 bits).

Description

glutBitmapWidth returns the width in pixels of a bitmap character in a supported bitmap font. While the width of characters in a font may vary (though fixed width fonts do not vary), the maximum height characteristics of a particular font are fixed.

10.3 glutStrokeCharacter

glutStrokeCharacter renders a stroke character using OpenGL.

Usage

void glutStrokeCharacter(void *font, int character);

font Stroke font to use.

character Character to render (not confined to 8 bits).

Description

Without using any display lists, glutStrokeCharacter renders the character in the named stroke font. The available fonts are:

- GLUT_STROKE_ROMAN A proportionally spaced Roman Simplex font for ASCII characters 32 through 127. The maximum top character in the font is 119.05 units; the bottom descends 33.33 units.
- GLUT_STROKE_MONO_ROMAN A mono-spaced spaced Roman Simplex font (same characters as GLUT_STROKE_ROMAN) for ASCII characters 32 through 127. The maximum top character in the font is 119.05 units; the bottom descends 33.33 units. Each character is 104.76 units wide.

Rendering a nonexistent character has no effect. A glTranslatef is used to translate the current model view matrix to advance the width of the character.

10.4 glutStrokeWidth

glutStrokeWidth returns the width of a stroke character.

Usage

```
int glutStrokeWidth(GLUTstrokeFont font, int character);
```

font Stroke font to use.

character Character to return width of (not confined to 8 bits).

Description

glutStrokeWidth returns the width in pixels of a stroke character in a supported stroke font. While the width of characters in a font may vary (though fixed width fonts do not vary), the maximum height characteristics of a particular font are fixed.

11 Geometric Object Rendering

GLUT includes a number of routines for generating easily recognizable 3D geometric objects. These routines reflect functionality available in the aux toolkit described in the *OpenGL Programmer's Guide*

and are included in GLUT to allow the construction of simple GLUT programs that render recognizable objects. These routines can be implemented as pure OpenGL rendering routines. The routines do *not* generate display lists for the objects they create.

The routines generate normals appropriate for lighting but do not generate texture coordinates (except for the teapot).

11.1 glutSolidSphere, glutWireSphere

glutSolidSphere and glutWireSphere render a solid or wireframe sphere respectively.

Usage

radius The radius of the sphere.

slices The number of subdivisions around the Z axis (similar to lines of longitude).

stacks The number of subdivisions along the Z axis (similar to lines of latitude).

Description

Renders a sphere centered at the modeling coordinates origin of the specified radius. The sphere is subdivided around the Z axis into slices and along the Z axis into stacks.

11.2 glutSolidCube, glutWireCube

glutSolidCube and glutWireCube render a solid or wireframe cube respectively.

36

11.3 glutSolidCone, glutWireCone

Usage

```
void glutSolidCube(GLdouble size);
void glutWireCube(GLdouble size);
```

size Length of each edge.

Description

glutSolidCube and glutWireCube render a solid or wireframe cube respectively. The cube is centered at the modeling coordinates origin with sides of length size.

11.3 glutSolidCone, glutWireCone

glutSolidCone and glutWireCone render a solid or wireframe cone respectively.

Usage

base The radius of the base of the cone.

height The height of the cone.

slices The number of subdivisions around the Z axis.

stacks The number of subdivisions along the Z axis.

Description

glutSolidCone and glutWireCone render a solid or wireframe cone respectively oriented along the Z axis. The base of the cone is placed at Z = 0, and the top at Z = height. The cone is subdivided around the Z axis into slices, and along the Z axis into stacks.

11.4 glutSolidTorus, glutWireTorus

glutSolidTorus and glutWireTorus render a solid or wireframe torus (doughnut) respectively.

Usage

innerRadius Inner radius of the torus.

outerRadius Outer radius of the torus.

nsides Number of sides for each radial section.

rings Number of radial divisions for the torus.

Description

glutSolidTorus and glutWireTorus render a solid or wireframe torus (doughnut) respectively centered at the modeling coordinates origin whose axis is aligned with the Z axis.

11.5 glutSolidDodecahedron, glutWireDodecahedron

glutSolidDodecahedron and glutWireDodecahedron render a solid or wireframe dodecahedron (12-sided regular solid) respectively.

Usage

```
void glutSolidDodecahedron(void);
void glutWireDodecahedron(void);
```

Description

glutSolidDodecahedron and glutWireDodecahedron render a solid or wireframe dodecahedron respectively centered at the modeling coordinates origin with a radius of $\sqrt{3}$.

11.6 glutSolidOctahedron, glutWireOctahedron

glutSolidOctahedron and glutWireOctahedron render a solid or wireframe octahedron (8-sided regular solid) respectively.

Usage

```
void glutSolidOctahedron(void);
void glutWireOctahedron(void);
```

Description

glutSolidOctahedron and glutWireOctahedron render a solid or wireframe octahedron respectively centered at the modeling coordinates origin with a radius of 1.0.

11.7 glutSolidTetrahedron, glutWireTetrahedron

glutSolidTetrahedron and glutWireTetrahedron render a solid or wireframe tetrahedron (4-sided regular solid) respectively.

Usage

```
void glutSolidTetrahedron(void);
void glutWireTetrahedron(void);
```

Description

glutSolidTetrahedron and glutWireTetrahedron render a solid or wireframe tetrahedron respectively centered at the modeling coordinates origin with a radius of $\sqrt{3}$.

11.8 glutSolidIcosahedron, glutWireIcosahedron

glutSolidIcosahedron and glutWireIcosahedron render a solid or wireframe icosahedron (20sided regular solid) respectively.

11.9 glutSolidTeapot, glutWireTeapot

Usage

```
void glutSolidIcosahedron(void);
void glutWireIcosahedron(void);
```

Description

glutSolidIcosahedron and glutWireIcosahedron render a solid or wireframe icosahedron respectively. The icosahedron is centered at the modeling coordinates origin and has a radius of 1.0.

11.9 glutSolidTeapot, glutWireTeapot

glutSolidTeapot and glutWireTeapot render a solid or wireframe teapot¹ respectively.

Usage

```
void glutSolidTeapot(GLdouble size);
void glutWireTeapot(GLdouble size);
```

size Relative size of the teapot.

Description

glutSolidTeapot and glutWireTeapot render a solid or wireframe teapot respectively. Both surface normals and texture coordinates for the teapot are generated. The teapot is generated with OpenGL evaluators.

12 Usage Advice

There are a number of points to keep in mind when writing GLUT programs. Some of these are strong recommendations, others simply hints and tips.

- Do not change state that will affect the way a window will be drawn in a window's display callback. Your display callbacks should be idempotent.
- If you need to redisplay a window, instead of rendering in whatever callback you happen to be in, call glutPostRedisplay (or glutPostRedisplay for overlays). As a general rule, the only code that renders directly to the screen should be in called from display callbacks; other types of callbacks should not be rendering to the screen.
- If you use an idle callback to control your animation, use the visibility callbacks to determine when the window is fully obscured or iconified to determine when not to waste processor time rendering.
- Neither GLUT nor the window system automatically reshape sub-windows. If subwindows should be reshaped to reflect a reshaping of the top-level window, the GLUT program is responsible for doing this.
- Avoid using color index mode if possible. The RGBA color model is more functional, and it is less likely to cause colormap swapping effects.
- Do not call any GLUT routine that affects the *current window* or *current menu* if there is no *current window* or *current menu* defined. This can be the case at initialization time (before any windows or menus have been created) or if your destroy the *current window* or *current menu*. GLUT implementations are not obliged to generate a warning because doing so would slow down the operation of every such routine to first make sure there was a *current window* or *current menu*.

¹Yes, the *classic* computer graphics teapot modeled by Martin Newell in 1975 [3].

- For most callbacks, the *current window* and/or *current menu* is set appropriately at the time of the callback. Timer and idle callbacks are exceptions. If your application uses multiple windows or menus, make sure you explicitly you set the *current window* or *menu* appropriately using glutSetWindow or glutSetMenu in the idle and timer callbacks.
- If you register a single function as a callback routine for multiple windows, you can call glutGetWindow within the callback to determine what window generated the callback. Likewise, glutGetMenu can be called to determine what menu.
- By default, timer and idle callbacks may be called while a pop-up menu is active. On slow machines, slow rendering in an idle callback may compromise menu performance. Also, it may be desirable for motion to stop immediately when a menu is triggered. In this case, use the menu entry/exit callback set with glutMenuStateFunc to track the usage of pop-up menus.
- Do not select for more input callbacks than you actually need. For example, if you do not need motion or passive motion callbacks, disable them by passing NULL to their callback register functions. Disabling input callbacks allows the GLUT implementation to limit the window system input events that must be processed.
- Not every OpenGL implementation supports the same range of frame buffer capabilities, though minimum requirements for frame buffer capabilities do exist. If glutCreateWindow or glutCreateSubWindow are called with an *initial display mode* not supported by the OpenGL implementation, a fatal error will be generated with an explanatory message. To avoid this, glutGet(GLUT_DISPLAY_MODE_POSSIBLE) should be called to determine if the *initial display mode* is supported by the OpenGL implementation.
- The Backspace, Delete, and Escape keys generate ASCII characters, so detect these key presses with the glutKeyboardFunc callback, not with the glutSpecialFunc callback.
- Keep in mind that when a window is damaged, you should assume *all* of the ancillary buffers are damaged and redraw them all.
- Keep in mind that after a glutSwapBuffers, you should assume the state of the back buffer becomes undefined.
- If not using glutSwapBuffers for double buffered animation, remember to use glFlush to make sure rendering requests are dispatched to the frame buffer. While many OpenGL implementations will automatically flush pending commands, this is specifically not mandated.
- Remember that it is illegal to create or destroy menus or change, add, or remove menu items while a menu (and any cascaded sub-menus) are in use (that is, "popped up"). Use the menu status callback to know when to avoid menu manipulation.
- It is more efficient to use glutHideOverlay and glutShowOverlay to control the display state of a window's overlay instead of removing and re-establishing an overlay every time an overlay is needed.
- Few workstations have support for multiple simultaneously installed overlay colormaps. For this reason, if an overlay is cleared or otherwise not be used, it is best to hide it using glutHideOverlay to avoid other windows with active overlays from being displayed with the wrong colormap. If your application uses multiple overlays, use glutCopyColormap to promote colormap sharing.
- If you are encountering GLUT warnings or fatal errors in your programs, try setting a debugger breakpoint in __glutWarning or __glutFatalError (though these names are potentially implementation dependent) to determine where within your program the error occurred.
- GLUT has no special routine for exiting the program. GLUT programs should use ANSI C's exit routine. If a program needs to perform special operations before quitting the program, use the ANSI C onexit routine to register exit callbacks. GLUT will exit the program unilaterally when fatal errors occur or when the window system requests the program to terminate. For this reason, avoid calling any GLUT routines within an exit callback.

• Definitely, definitely, use the -gldebug option to look for OpenGL errors when OpenGL rendering does not appear to be operating properly. OpenGL errors are only reported if you explicitly look for them!

13 FORTRAN Binding

All GLUT functionality is available through the GLUT FORTRAN API. The GLUT FORTRAN binding is intended to be used in conjunction with the OpenGL and GLU FORTRAN APIs.

A FORTRAN routine using GLUT routines should include the GLUT FORTRAN header file. While this is potentially system dependent, on Unix systems this is normally done by including after the SUBROUTINE, FUNCTION, or PROGRAM line:

```
#include "GL/fglut.h"
```

Though the FORTRAN 77 specification differentiates identifiers by their first six characters only, the GLUT FORTRAN binding (and the OpenGL and GLU FORTRAN bindings) assume identifiers are not limited to 6 characters.

The FORTRAN GLUT binding library archive is typically named libfglut.a on Unix systems. FOR-TRAN GLUT programs need to link with the system's OpenGL and GLUT libraries and the respective Fortran binding libraries (and any libraries these libraries potentially depend on). A set of window system dependent libraries may also be necessary for linking GLUT programs. For example, programs using the X11 GLUT implementation typically need to link with Xlib, the X extension library, possibly the X Input extension library, the X miscellaneous utilities library, and the math library. An example X11/Unix compile line for a GLUT FOR-TRAN program would look like:

```
f77 -o foo foo.c -lfglut -lglut -lfGLU -lGLU -lfGL -lGL \
-lXmu -lXi -lXext -lX11 -lm
```

13.1 Names for the FORTRAN GLUT Binding

Allowing for FORTRAN's case-insensitivity, the GLUT FORTRAN binding constant and routine names are the same as the C binding's names.

The OpenGL Architectural Review Board (ARB) official OpenGL FORTRAN API prefixes every routine and constant with the letter F. The justification was to avoid name space collisions with the C names in anachronistic compilers. Nearly all modern FORTRAN compilers avoid these name space clashes via other means (underbar suffixing of FORTRAN routines is used by most Unix FORTRAN compilers).

The GLUT FORTRAN API does *not* use such prefixing conventions because of the documentation and coding confusion introduced by such prefixes. The confusion is heightened by FORTRAN's default implicit variable initialization so programmers may realize the lack of a constant prefix as a result of a run-time error. The confusion introduced to support the prefixes was not deemed worthwhile simply to support anachronistic compliers.

13.2 Font Naming Caveat

Because GLUT fonts are compiled directly into GLUT programs as data, and programs should only have the fonts compiled into them that they use, GLUT font names like GLUT_BITMAP_TIMES_ROMAN_24 are really symbols so the linker should only pull in used fonts.

Unfortunately, because some supposedly modern FORTRAN compilers link declared but unused data EXTERNALs, "GL/fglut.h" does not explicitly declare EXTERNAL the GLUT font symbols. Declaring the GLUT font symbols as EXTERNAL risks forcing every GLUT FORTRAN program to contain the data for every GLUT font. GLUT Fortran programmers should explicitly declare EXTERNAL the GLUT fonts they use. Example:

```
SUBROUTINE PRINTA
#include "GL/fglut.h"
EXTERNAL GLUT_BITMAP_TIMES_ROMAN_24
CALL glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, 65)
END
```

13.3 NULL Callback

FORTRAN does not support passing NULL as a callback parameter the way ANSI C does. For this reason, GLUTNULL is used in place of NULL in GLUT FORTRAN programs to indicate a NULL callback.

14 Implementation Issues

While this specification is primarily intended to describe the GLUT API and not its implementation, the section describes implementation issues that are likely to help both GLUT implementors properly implement GLUT and provide GLUT programmers with information to better utilize GLUT.

14.1 Name Space Conventions

The GLUT implementation should have a well-defined name space for both exported symbols and visible, but not purposefully exported symbols. All exported functions are prefixed by glut. All exported macro definitions are prefixed by GLUT_. No data symbols are exported. All internal symbols that might be user-visible but not intended to be exported should be prefixed by _glut. Users of the GLUT API should *not* use any _glut prefixed symbols.

14.2 Modular Implementation

It is often the case that windowing libraries tend to result in large, bulky programs because a large measure of "dynamically dead" code is linked into the programs because it can not be determined at link time that the program will never require (that is, execute) the code. A consideration (not a primary one though) in GLUT's API design is make the API modular enough that programs using a limited subset of GLUT's API can minimize the portion of the GLUT library implementation required. This does assume the implementation of GLUT is structured to take advantage of the API's modularity.

A good implementation can be structured so significant chunks of code for color index colormap management, non-standard device support (Spaceball, dial & button box, and tablet), overlay management, pop-up menus, miscellaneous window management routines (pop, push, show, hide, full screen, iconify), geometric shape rendering, and font rendering only need to be pulled into GLUT programs when the interface to this functionality is explicitly used by the GLUT program.

14.3 Error Checking and Reporting

How errors and warnings about improper GLUT usage are reported to GLUT programs is implementation dependent. The recommended behavior in the case of an error is to output a message and exit. In the case of a warning, the recommended behavior is to output a message and continue. All improper uses of the GLUT interface do not need to be caught or reported. What conditions are caught or reported should be based on how expensive the condition is to check for. For example, an implementation may not check every glutSetWindow call to determine if the window identifier is valid.

The run-time overhead of error checking for a very common operation may outweight the benefit of clean error reporting. This trade-off is left for the implementor to make. The implementor should also consider the difficulty of diagnosing the improper usage without a message being output. For example, if a GLUT program attempts to create a menu while a menu is in use (improper usage!), this warrants a message because this improper usage may often be benign, allowing the bug to easily go unnoticed.

14.4 Avoid Unspecified GLUT Usage Restrictions

GLUT implementations should be careful to not limit the conditions under which GLUT routines may be called. GLUT implementations are expected to be resilient when GLUT programs call GLUT routines with defined behavior at "unexpected" times. For example, a program should be permitted to destroy the *current window* from within a display callback (assuming the user does not then call GLUT routines requiring a *current window*). This means after dispatching callbacks, a GLUT implementation should be "defensive" about how the program might have used manipulated GLUT state during the callback.

A GLUT State

This appendix specifies precisely what programmer visible state GLUT maintains. There are three categories of programmer visible state that GLUT maintains: global, window, and menu. The window and menu state categories are maintained for each created window or menu. Additional overlay-related window state is maintained when an overlay is established for a window for the lifetime of the overlay.

The tables below name each element of state, define its type, specify what GLUT API entry points set or change the state (if possible), specify what GLUT API entry point or glutGet, glutDeviceGet, or glutLayerGet state constant is used to get the state (if possible), and how the state is initially set. For details of how any API entry point operates on the specified state, see the routine's official description. Footnotes for each category of state indicate additional caveats to the element of state.

A.1 Types of State

These types are used to specify GLUT's programmer visible state:

Bitmask A group of boolean bits.

Boolean True or false.

- **Callback** A handle to a user-supplied routine invoked when the given callback is triggered (or NULL which is the default callback).
- ColorCell Red, green, and blue color component triple, an array of which makes a colormap.

Cursor A GLUT cursor name.

Integer An integer value.

Layer Either normal plane or overlay.

- **MenuItem** Either a menu entry or a submenu trigger. Both subtypes contain of a *String* name. A menu entry has an *Integer* value. A submenu cascade has an *Integer* menu name naming its associated submenu.
- MenuState Either in use or not in use.
- **Stacking** An ordering for top-level windows and sub-windows having the same parent. Higher windows obscure lower windows.

State One of shown, hidden, or iconified.

String A string of ASCII characters.

Timer A triple of a timer *Callback*, an *Integer* callback parameter, and a time in milliseconds (that expires in real time).

A.2 Global State

There are two types of global state: program controlled state which can be modified directly or indirectly by the program, and fixed system dependent state.

A.3 Window State

A.2.1 Program Controlled State

Name	Туре	Set/Change	Get	Initial
currentWindow	Integer	glutSetWindow (1)	glutGetWindow	0
currentMenu	Integer	glutSetMenu (2)	glutGetMenu	0
initWindowX	Integer	glutInitWindowPosition	GLUT_INIT_WINDOW_X	-1
initWindowY	Integer	glutInitWindowPosition	GLUT_INIT_WINDOW_Y	-1
initWindowWidth	Integer	glutInitWindowSize	GLUT_INIT_WINDOW_WIDTH	300
initWindowHeight	Integer	glutInitWindowSize	GLUT_INIT_WINDOW_HEIGHT	300
initDisplayMode	Bitmask	glutInitDisplayMode	GLUT_INIT_DISPLAY_MODE	GLUT_RGB,
				GLUT_SINGLE,
				GLUT_DEPTH
idleCallback	Callback	glutIdleFunc	-	NULL
menuState	MenuState	-	(3)	NotInUse
menuStateCallback	Callback	glutMenuEntryFunc	-	NULL
timerList	list of Timer	glutTimerFunc	-	none

(1) The *currentWindow* is also changed implicitly by every window or menu callback (to the window triggering the callback) and the creation of a window (to the window being created).

(2) The *currentMenu* is also changed implicitly by every menu callback (to the menu triggering the callback) and the creation of a menu (to the menu being created).

(3) The menu state callback is triggered when the menuState changes.

Name	Туре	Get
screenWidth	Integer	GLUT_SCREEN_WIDTH
screenHeight	Integer	GLUT_SCREEN_HEIGHT
screenWidthMM	Integer	GLUT_SCREEN_WIDTH_MM
screenHeightMM	Integer	GLUT_SCREEN_HEIGHT_MM
hasKeyboard	Boolean	GLUT_HAS_KEYBOARD
hasMouse	Boolean	GLUT_HAS_MOUSE
hasSpaceball	Boolean	GLUT_HAS_SPACEBALL
hasDialAndButtonBox	Boolean	GLUT_HAS_DIAL_AND_BUTTON_BOX
hasTablet	Boolean	GLUT_HAS_TABLET
numMouseButtons	Integer	GLUT_NUM_MOUSE_BUTTONS
numSpaceballButtons	Integer	GLUT_NUM_SPACEBALL_BUTTONS
numButtonBoxButtons	Integer	GLUT_NUM_BUTTON_BOX_BUTTONS
numDials	Integer	GLUT_NUM_DIALS
numTabletButtons	Integer	GLUT_NUM_TABLET_BUTTONS

A.2.2 Fixed System Dependent State

A.3 Window State

For the purposes of listing the window state elements, window state is classified into three types: base state, frame buffer capability state, and layer state. The tags *top-level*, *sub-win*, and *cindex* indicate the table entry applies only to top-level windows, subwindows, or color index windows respectively.

A.3.1 Basic State

Name	Туре	Set/Change	Get	Initial
number	Integer	-	glutGetWindow	top-level: glutCreateWindow (1)
				sub-win: glutCreateSubWindow(1)
х	Integer	glutPositionWindow	GLUT_WINDOW_X	top-level: initWindowX (2)
				sub-win: glutCreateSubWindow
у	Integer	glutPositionWindow	GLUT_WINDOW_Y	top-level: initWindowY (3)
-	-	-		sub-win: glutCreateSubWindow
width	Integer	glutReshapeWindow	GLUT_WINDOW_WIDTH	top-level: initWindowWidth (4)
	C	0 1		sub-win: glutCreateSubWindow
height	Integer	glutReshapeWindow	GLUT_WINDOW_HEIGHT	top-level: initWindowHeight (5)
6		8		sub-win: glutCreateSubWindow
top-level: fullScreen	Boolean	glutFullScreen		False
		glutPositionWindow		
		glutReshapeWindow(6)		
cursor	Cursor	glutSetCursor	GLUT_WINDOW_CURSOR	GLUT_CURSOR_INHERIT
stacking	Stacking	glutPopWindow	-	top
sucking	Stucking	glutPushWindow		юр
displayState	State (7)	glutShowWindow (8)	-	shown
displaybate	State (7)	glutHideWindow		310 w 11
		glutIconifyWindow		
visibility	Visibility	(9)	(10)	undefined
redisplay	Boolean	glutPostRedisplay (11)	-	False
top-level: windowTitle	String	glutWindowTitle		glutCreateWindow
top-level: iconTitle	String	glutIconTitle	-	glutCreateWindow
displayCallback	Callback	glutDisplayFunc	-	NULL (12)
reshapeCallback	Callback	glutReshapeFunc	-	NULL (12) NULL (13)
keyboardCallback	Callback	glutKeyboardFunc	-	NULL
mouseCallback	Callback	glutMouseFunc	-	NULL
	Callback	glutMotionFunc	-	NULL
motionCallback		8	-	
passiveMotionCallback	Callback	glutPassiveMotionFunc	-	NULL
specialCallback	Callback	glutSpecialFunc	-	NULL
spaceballMotionCallback	Callback	glutSpaceballMotionFunc	-	NULL
spaceballRotateCallback	Callback	glutSpaceballRotateFunc	-	NULL
spaceballButtonCallback	Callback	glutSpaceballButtonFunc	-	NULL
buttonBoxCallback	Callback	glutButtonBoxFunc	-	NULL
dialsCallback	Callback	glutDialsFunc	-	NULL
tabletMotionCallback	Callback	glutTabletMotionFunc	-	NULL
tabletButtonCallback	Callback	glutTabletButtonFunc	-	NULL
visibilityCallback	Callback	glutVisibilityFunc	-	NULL
entryCallback	Callback	glutEntryFunc	-	NULL
cindex: colormap	array of	glutSetColor	glutGetColor	undefined
	ColorCell	glutCopyColormap		
windowParent	Integer	-	GLUT_WINDOW_PARENT	top-level: 0
				sub-win: (14)
numChildren	Integer	glutCreateSubWindow	GLUT_NUM_CHILDREN	0
		glutDestroyWindow		
leftMenu	Integer	glutAttachMenu	-	0
		glutDetachMenu		
middleMenu	Integer	glutAttachMenu	-	0
		glutDetachMenu		
rightMenu	Integer	glutAttachMenu	-	0
		glutDetachMenu		

(1) Assigned dynamically from unassigned window numbers greater than zero.

(2) If *initWindowX* is greater or equal to zero *and initWindowY* is greater or equal to zero then *initWindowX*, else window location left to window system to decide.

(3) If *initWindowY* is greater or equal to zero *and initWindowX* is greater or equal to zero then *initWindowY*, else window location left to window system to decide.

(4) If *initWindowWidth* is greater than zero *and initWindowHeight* is greater than zero the *initWindowWidth*, else window size left to window system to decide.

(5) If *initWindowHeight* is greater than zero *and initWindowWidth* is greater than zero then *initWindowHeight*, else window size left to window system to decide.

(6) glutFullScreen sets to true; glutPositionWindow and glutReshapeWindow set to false.

(7) Subwindows can not be iconified.

(8) Window system events can also change the displayState.

A.3 Window State

- (9) Visibility of a window can change for window system dependent reason, for example, a new window may occlude the window. glutPopWindow and glutPushWindow can affect window visibility as a side effect.
- (10) The visibility callback set by glutVisibilityFunc allows the visibility state to be tracked.
- (11) The redisplay state can be explicitly enabled by glutRedisplayFunc or implicitly in response to normal plane redisplay events from the window system.
- (12) A window's displayCallback must be registered before the first display callback would be triggered (or the program is terminated).
- (13) Instead of being a no-op as most NULL callbacks are, a NULL *reshapeCallback* sets the OpenGL viewport to render into the complete window, i.e., glViewport(0,0,width, height).
- (14) Determined by *currentWindow* at glutCreateSubWindow time.

A.3.2 Frame Buffer Capability State

Name	Туре	Get
Total number of bits in color buffer	Integer	GLUT_WINDOW_BUFFER_SIZE
Number of bits in stencil buffer	Integer	GLUT_WINDOW_STENCIL_SIZE
Number of bits in depth buffer	Integer	GLUT_WINDOW_DEPTH_SIZE
Number of bits of red stored in color buffer	Integer	GLUT_WINDOW_RED_SIZE
Number of bits of green stored in color buffer	Integer	GLUT_WINDOW_GREEN_SIZE
Number of bits of blue stored in color buffer	Integer	GLUT_WINDOW_BLUE_SIZE
Number of bits of alpha stored in color buffer	Integer	GLUT_WINDOW_ALPHA_SIZE
Number of bits of red stored in accumulation buffer	Integer	GLUT_WINDOW_ACCUM_RED_SIZE
Number of bits of green stored in accumulation buffer	Integer	GLUT_WINDOW_ACCUM_GREEN_SIZE
Number of bits of blue stored in accumulation buffer	Integer	GLUT_WINDOW_ACCUM_BLUE_SIZE
Number of bits of alpha stored in accumulation buffer	Integer	GLUT_WINDOW_ACCUM_ALPHA_SIZE
Color index colormap size	Integer	GLUT_WINDOW_COLORMAP_SIZE
If double buffered	Boolean	GLUT_WINDOW_DOUBLEBUFFER
If RGBA color model	Boolean	GLUT_WINDOW_RGBA
If stereo	Boolean	GLUT_WINDOW_STEREO
Number of samples for multisampling	Integer	GLUT_WINDOW_MULTISAMPLE

A window's (normal plane) frame buffer capability state is derived from the global initDisplayMode state at the window's creation. A window's frame buffer capabilities can not be changed.

A.3.3 Layer State

Name	Туре	Set/Change	Get	Initial
hasOverlay	Boolean	glutEstablishOverlay	GLUT_HAS_OVERLAY	False
		glutRemoveOverlay		
overlayPossible	Boolean	(1)	GLUT_OVERLAY_POSSIBLE	False
layerInUse	Layer	glutUseLayer (2)	GLUT_LAYER_IN_USE	normal plane
cindex: transparentIndex	Integer	-	GLUT_TRANSPARENT_INDEX	(3)
overlayRedisplay	Boolean	glutPostOverlayRedisplay (4)	-	False
overlayDisplayCallback	Callback	glutOverlayDisplayFunc	-	NULL
overlayDisplayState	State	glutShowOverlay	-	shown
		glutHideOverlay		
normalDamaged	Boolean	(5)	GLUT_NORMAL_DAMAGED	False
overlayDamaged	Boolean	(6)	GLUT_OVERLAY_DAMAGED	False

(1) Whether an overlay is possible is based on the *initDisplayMode* state and the frame buffer capability state of the window.

(2) The *layerInUse* is implicitly set to overlay after glutEstablishOverlay; likewise, glutRemoveOverlay resets the state to normal plane.

(3) The *transparentIndex* is set when a color index overlay is established. It cannot be set; it may change if the overlay is re-established. When no overlay is in use or if the overlay is not color index, the *transparentIndex* is -1.

- (4) The overlayRedisplay state can be explicitly enabled by glutPostOverlayRedisplay or implicitly in response to overlay redisplay events from the window system.
- (5) Set when the window system reports a region of the window's normal plane is undefined (for example, damaged by another window moving or being initially shown). The specifics of when damage occurs are left to the window system to determine. The window's *redisplay* state is always set true when damage occurs. *normalDamaged* is cleared whenever the window's display callback returns.
- (6) Set when the window system reports a region of the window's overlay plane is undefined (for example, damaged by another window moving or being initially shown). The specifics of when damage occurs are left to the window system to determine. The damage may occur independent from damage to the window's normal plane. The window's *redisplay* state is always set true when damage occurs. *normalDamaged* is cleared whenever the window's display callback returns.

When an overlay is established, *overlay* frame buffer capability state is maintained as described in Section A.3.2. The *layerInUse* determines whether glutGet returns normal plane or overlay state when an overlay is established.

A.4 Menu State

Name	Туре	Set/Change	Get	Initial
number	Integer	-	glutSetMenu	top-level: glutCreateMenu (1)
select	Callback	-	-	glutCreateMenu
items	list of MenuItem	-	-	-
numItems	Integer	-	GLUT_MENU_NUM_ITEMS	0

(1) Assigned dynamically from unassigned window numbers greater than zero.

B glut.h ANSI C Header File

```
#ifndef __glut_h__
 1
 2 #define __glut_h__
 3
 4 /* Copyright (c) Mark J. Kilgard, 1994, 1995, 1996. */
 5
 б
   /* This program is freely distributable without licensing fees and is
7
       provided without guarantee or warrantee expressed or implied. This
 8
      program is -not- in the public domain. */
 9
10 #include <GL/ql.h>
   #include <GL/glu.h>
11
12
13 #ifdef __cplusplus
14 extern "C" {
15 #endif
16
17
   /*
   * GLUT API revision history:
18
19
20
    * GLUT_API_VERSION is updated to reflect incompatible GLUT
21
    * API changes (interface changes, semantic changes, deletions,
    * or additions).
2.2
23
    * GLUT_API_VERSION=1 First public release of GLUT. 11/29/94
24
25
    * GLUT_API_VERSION=2 Added support for OpenGL/GLX multisampling,
26
27
     * extension. Supports new input devices like tablet, dial and button
28
    * box, and Spaceball. Easy to query OpenGL extensions.
29
30
    * GLUT_API_VERSION=3 glutMenuStatus added.
31
    * /
32
33 #ifndef GLUT_API_VERSION /* allow this to be overriden */
34 #define GLUT_API_VERSION
                                           3
35 #endif
36
37
   /*
   * GLUT implementation revision history:
38
39
    * GLUT_XLIB_IMPLEMENTATION is updated to reflect both GLUT
40
    \star API revisions and implementation revisions (ie, bug fixes).
41
42
    * GLUT_XLIB_IMPLEMENTATION=1 mjk's first public release of
43
44
    * GLUT Xlib-based implementation. 11/29/94
45
46
     * GLUT_XLIB_IMPLEMENTATION=2 mjk's second public release of
    * GLUT Xlib-based implementation providing GLUT version 2
47
    interfaces.
48
49
    * GLUT_XLIB_IMPLEMENTATION=3 mjk's GLUT 2.2 images. 4/17/95
50
51
52
    * GLUT_XLIB_IMPLEMENTATION=4 mjk's GLUT 2.3 images. 6/?/95
53
    * GLUT_XLIB_IMPLEMENTATION=5 mjk's GLUT 3.0 images. 10/?/95
54
55
56
    * GLUT_XLIB_IMPLEMENTATION=6 mjk's GLUT 3.1
57
    * /
58 #ifndef GLUT_XLIB_IMPLEMENTATION /* allow this to be overriden */
    #define GLUT_XLIB_IMPLEMENTATION
59
                                           б
60 #endif
61
62 /* display mode bit masks */
63 #define GLUT_RGB
                                           0
64 #define GLUT_RGBA
                                           GLUT_RGB
65 #define GLUT_INDEX
                                           1
66 #define GLUT_SINGLE
                                           0
```

67	#define	GLUT_DOUBLE	2
68		 GLUT_ACCUM	4
69		GLUT_ALPHA	8
		—	
70		GLUT_DEPTH	16
71		GLUT_STENCIL	32
72		JT_API_VERSION >= 2)	
73	#define	GLUT_MULTISAMPLE	128
74	#define	GLUT_STEREO	256
75	#endif		
76		JT_API_VERSION >= 3)	
77		GLUT_LUMINANCE	512
		GEOI_LOMINANCE	51Z
78	#endif		
79			
80	/* mouse	e buttons */	
81	#define	GLUT_LEFT_BUTTON	0
82	#define	GLUT_MIDDLE_BUTTON	1
83		GLUT_RIGHT_BUTTON	2
84	#401 1110	0101_n10n1_b0110n	-
	/+	button college state */	
		e button callback state */	0
86		GLUT_DOWN	0
87	#define	GLUT_UP	1
88			
89	#if (GLU	JT_API_VERSION >= 2)	
90	/* funct	tion keys */	
91		GLUT_KEY_F1	1
92		GLUT_KEY_F2	2
93		GLUT_KEY_F3	3
94		GLUT_KEY_F4	4
95	#define	GLUT_KEY_F5	5
96	#define	GLUT_KEY_F6	6
97	#define	GLUT_KEY_F7	7
98		GLUT_KEY_F8	8
99		GLUT_KEY_F9	9
100		GLUT_KEY_F10	10
		GLUT_KEY_F11	11
102	#define	GLUT_KEY_F12	12
103	/* dired	ctional keys */	
104	#define	GLUT_KEY_LEFT	100
105	#define	GLUT_KEY_UP	101
		GLUT_KEY_RIGHT	102
107		GLUT KEY DOWN	102
108		GLUT_KEY_PAGE_UP	104
		GLUT_KEY_PAGE_DOWN	105
110	#define	GLUT_KEY_HOME	106
111	#define	GLUT_KEY_END	107
112	#define	GLUT_KEY_INSERT	108
113	#endif		
114	() CHIGEL		
	/*	·/arrit mallhaml state */	
115		//exit callback state */	
116		GLUT_LEFT	0
117	#define	GLUT_ENTERED	1
118			
119	/* menu	usage callback state */	
120	#define	GLUT_MENU_NOT_IN_USE	0
121		GLUT_MENU_IN_USE	1
	#del Inc	GTO1_HENO_TH_OPE	1
122	/+! -! !!		
123		pility callback state */	
124		GLUT_NOT_VISIBLE	0
125	#define	GLUT_VISIBLE	1
126			
127	/* color	r index component selection value	es */
128		GLUT_RED	0
129		GLUT_GREEN	1
130	#aerine	GLUT_BLUE	2
131			
132	-	rs for use */	
133	#define	GLUT_NORMAL	0
134	#define	GLUT_OVERLAY	1

50

```
135
136 /* stroke font opaque addresses (use constants instead in source code) */
137 extern void *glutStrokeRoman;
138 extern void *glutStrokeMonoRoman;
129
140 /* stroke font constants (use these in GLUT program) */
141 #define GLUT STROKE ROMAN
                                          (&glutStrokeRoman)
142 #define GLUT_STROKE_MONO_ROMAN
                                          (&glutStrokeMonoRoman)
143
144 /* bitmap font opaque addresses (use constants instead in source code) */
145 extern void *glutBitmap9By15;
146 extern void *glutBitmap8By13;
147 extern void *glutBitmapTimesRoman10;
148 extern void *glutBitmapTimesRoman24;
149 extern void *glutBitmapHelvetica10;
150 extern void *glutBitmapHelvetical2;
151 extern void *glutBitmapHelvetica18;
152
153
    /* bitmap font constants (use these in GLUT program) */
154 #define GLUT_BITMAP_9_BY_15 (&glutBitmap9By15)
155 #define GLUT_BITMAP_8_BY_13
                                           (&glutBitmap8By13)
156 #define GLUT_BITMAP_TIMES_ROMAN_10
                                          (&glutBitmapTimesRoman10)
157 #define GLUT_BITMAP_TIMES_ROMAN_24
                                          (&glutBitmapTimesRoman24)
158 #if (GLUT_API_VERSION >= 3)
159 #define GLUT_BITMAP_HELVETICA_10
                                          (&glutBitmapHelvetica10)
160 #define GLUT_BITMAP_HELVETICA_12
                                           (&glutBitmapHelvetica12)
161 #define GLUT_BITMAP_HELVETICA_18
                                           (&glutBitmapHelvetica18)
162 #endif
163
164 /* glutGet parameters */
165 #define GLUT_WINDOW_X
                                           100
166 #define GLUT_WINDOW_Y
                                           101
167 #define GLUT_WINDOW_WIDTH
                                           102
168 #define GLUT_WINDOW_HEIGHT
                                           103
169 #define GLUT WINDOW BUFFER SIZE
                                          104
170 #define GLUT_WINDOW_STENCIL_SIZE
                                          105
171 #define GLUT_WINDOW_DEPTH_SIZE
                                           106
172 #define GLUT_WINDOW_RED_SIZE
                                           107
173 #define GLUT_WINDOW_GREEN_SIZE
                                           108
174 #define GLUT_WINDOW_BLUE_SIZE
                                           109
175 #define GLUT_WINDOW_ALPHA_SIZE
                                          110
176 #define GLUT_WINDOW_ACCUM_RED_SIZE
                                           111
177 #define GLUT_WINDOW_ACCUM_GREEN_SIZE
                                           112
178 #define GLUT_WINDOW_ACCUM_BLUE_SIZE
                                           113
179 #define GLUT_WINDOW_ACCUM_ALPHA_SIZE
                                           114
180 #define GLUT_WINDOW_DOUBLEBUFFER
                                           115
181 #define GLUT_WINDOW_RGBA
                                           116
182 #define GLUT_WINDOW_PARENT
                                           117
183 #define GLUT_WINDOW_NUM_CHILDREN
                                           118
184 #define GLUT_WINDOW_COLORMAP_SIZE
                                           119
185 #if (GLUT_API_VERSION >= 2)
186 #define GLUT_WINDOW_NUM_SAMPLES
                                           120
187 #define GLUT_WINDOW_STEREO
                                           121
188 #endif
189 #if (GLUT_API_VERSION >= 3)
190 #define GLUT_WINDOW_CURSOR
                                           122
191 #endif
192 #define GLUT_SCREEN_WIDTH
                                           200
193 #define GLUT_SCREEN_HEIGHT
                                           201
194 #define GLUT_SCREEN_WIDTH_MM
                                           202
195 #define GLUT_SCREEN_HEIGHT_MM
                                           203
196 #define GLUT_MENU_NUM_ITEMS
                                           300
197 #define GLUT_DISPLAY_MODE_POSSIBLE
                                           400
198 #define GLUT INIT WINDOW X
                                           500
199 #define GLUT_INIT_WINDOW_Y
                                           501
200 #define GLUT_INIT_WINDOW_WIDTH
                                           502
201
    #define GLUT_INIT_WINDOW_HEIGHT
                                           503
202 #define GLUT_INIT_DISPLAY_MODE
                                           504
```

B. GLUT.H ANSI C HEADER FILE

203 #if (GLUT API VERSION >= 2) 204 #define GLUT_ELAPSED_TIME 700 205 #endif 206 207 #if (GLUT_API_VERSION >= 2) 208 /* glutDeviceGet parameters */ 209 #define GLUT_HAS_KEYBOARD 600 210 #define GLUT_HAS_MOUSE 601 211 #define GLUT_HAS_SPACEBALL 602 212 #define GLUT_HAS_DIAL_AND_BUTTON_BOX 603 214 #define GLUT_NUM_MOUSE_BUTTONS 215 #define GLUT_NUM_MOUSE_BUTTONS 604 605 215 #define GLUT_NUM_SPACEBALL_BUTTONS 606 216 #define GLUT_NUM_BUTTON_BOX_BUTTONS 607 217 #define GLUT_NUM_DIALS 608 218 #define GLUT_NUM_TABLET_BUTTONS 609 219 #endif 220 221 #if (GLUT_API_VERSION >= 3) 222 /* glutLayerGet parameters */ 223 #define GLUT_OVERLAY_POSSIBLE 800 224 #define GLUT_LAYER_IN_USE 801 225 #define GLUT_HAS_OVERLAY 802 226 #define GLUT_TRANSPARENT_INDEX 803 227 #define GLUT_NORMAL_DAMAGED 804 228 #define GLUT_OVERLAY_DAMAGED 805 229 230 /* glutUseLayer parameters */ 231 #define GLUT_NORMAL 0 232 #define GLUT_OVERLAY 1 233 234 /* glutGetModifiers return mask */ 235 #define GLUT_ACTIVE_SHIFT 1 236 #define GLUT_ACTIVE_CTRL 2 237 #define GLUT_ACTIVE_ALT 4 238 239 /* glutSetCursor parameters */ 240 /* Basic arrows */ 241 #define GLUT_CURSOR_RIGHT_ARROW 0 242 #define GLUT_CURSOR_LEFT_ARROW 1 243 /* Symbolic cursor shapees */ 244 #define GLUT_CURSOR_INFO 2 245 #define GLUT_CURSOR_DESTROY 3 246 #define GLUT_CURSOR_HELP 4 247 #define GLUT_CURSOR_CYCLE 5 248 #define GLUT_CURSOR_SPRAY 6 249 #define GLUT_CURSOR_WAIT 7 250 #define GLUT_CURSOR_TEXT 8 251 #define GLUT_CURSOR_CROSSHAIR 9 252 /* Directional cursors */ 253 #define GLUT_CURSOR_UP_DOWN 10 254 #define GLUT_CURSOR_LEFT_RIGHT 11 255 /* Sizing cursors */ 256 #define GLUT_CURSOR_TOP_SIDE 12 257 #define GLUT_CURSOR_BOTTOM_SIDE 13 258 #define GLUT_CURSOR_LEFT_SIDE 14 259 #define GLUT CURSOR RIGHT SIDE 15 260 #define GLUT_CURSOR_TOP_LEFT_CORNER 16 261 #define GLUT_CURSOR_TOP_RIGHT_CORNER 17 262 #define GLUT_CURSOR_BOTTOM_RIGHT_CORNER 18 263 #define GLUT_CURSOR_BOTTOM_LEFT_CORNER 19 264 /* Inherit from parent window */ 265 #define GLUT_CURSOR_INHERIT 100 266 /* Blank cursor */ 267 #define GLUT_CURSOR_NONE 101 268 /* Fullscreen crosshair (if available) */ 269 #define GLUT_CURSOR_FULL_CROSSHAIR 102 270 #endif

52

```
271
272 /* GLUT initialization sub-API */
273 extern void glutInit(int *argcp, char **argv);
274 extern void glutInitDisplayMode(unsigned int mode);
275 extern void glutInitWindowPosition(int x, int y);
276 extern void glutInitWindowSize(int width, int height);
277 extern void glutMainLoop(void);
278
279
    /* GLUT window sub-api */
280 extern int glutCreateWindow(char *title);
281 extern int glutCreateSubWindow(int win, int x, int y, int width, int height);
282 extern void glutDestroyWindow(int win);
283 extern void glutPostRedisplay(void);
284 extern void glutSwapBuffers(void);
285 extern int glutGetWindow(void);
286 extern void glutSetWindow(int win);
287 extern void glutSetWindowTitle(char *title);
288 extern void glutSetIconTitle(char *title);
289 extern void glutPositionWindow(int x, int y);
290 extern void glutReshapeWindow(int width, int height);
291 extern void glutPopWindow(void);
292 extern void glutPushWindow(void);
293 extern void glutIconifyWindow(void);
294 extern void glutShowWindow(void);
295 extern void glutHideWindow(void);
296 #if (GLUT_API_VERSION >= 3)
297 extern void glutFullScreen(void);
298 extern void glutSetCursor(int cursor);
299
300 /* GLUT overlay sub-API */
301 extern void glutEstablishOverlay(void);
302 extern void glutRemoveOverlay(void);
303 extern void glutUseLayer(GLenum layer);
304 extern void glutPostOverlayRedisplay(void);
305 extern void glutShowOverlay(void);
306 extern void glutHideOverlay(void);
307 #endif
308
309 /* GLUT menu sub-API */
310 extern int glutCreateMenu(void (*)(int));
311 extern void glutDestroyMenu(int menu);
312 extern int glutGetMenu(void);
313 extern void glutSetMenu(int menu);
314 extern void glutAddMenuEntry(char *label, int value);
315 extern void glutAddSubMenu(char *label, int submenu);
316 extern void glutChangeToMenuEntry(int item, char *label, int value);
317 extern void glutChangeToSubMenu(int item, char *label, int submenu);
318 extern void glutRemoveMenuItem(int item);
319 extern void glutAttachMenu(int button);
320 extern void glutDetachMenu(int button);
321
322 /* GLUT callback sub-api */
323 extern void glutDisplayFunc(void (*)(void));
324 extern void glutReshapeFunc(void (*)(int width, int height));
325 extern void glutKeyboardFunc(void (*)(unsigned char key, int x, int y));
326 extern void glutMouseFunc(void (*)(int button, int state, int x, int y));
327 extern void glutMotionFunc(void (*)(int x, int y));
328 extern void glutPassiveMotionFunc(void (*)(int x, int y));
329 extern void glutEntryFunc(void (*)(int state));
330 extern void glutVisibilityFunc(void (*)(int state));
331 extern void glutIdleFunc(void (*)(void));
332 extern void glutTimerFunc(unsigned int millis, void (*)(int value), int value);
333 extern void glutMenuStateFunc(void (*)(int state));
334 #if (GLUT_API_VERSION >= 2)
335 extern void glutSpecialFunc(void (*)(int key, int x, int y));
336 extern void glutSpaceballMotionFunc(void (*)(int x, int y, int z));
337 extern void glutSpaceballRotateFunc(void (*)(int x, int y, int z));
338 extern void glutSpaceballButtonFunc(void (*)(int button, int state));
```

```
339 extern void glutButtonBoxFunc(void (*)(int button, int state));
340 extern void glutDialsFunc(void (*)(int dial, int value));
341 extern void glutTabletMotionFunc(void (*)(int x, int y));
342 extern void glutTabletButtonFunc(void (*)(int button, int state, int x, int y));
343 #if (GLUT_API_VERSION >= 3)
344 extern void glutMenuStatusFunc(void (*)(int status, int x, int y));
345 extern void glutOverlayDisplayFunc(void (*)(void));
346 #endif
347
    #endif
348
349 /* GLUT color index sub-api */
350 extern void glutSetColor(int, GLfloat red, GLfloat green, GLfloat blue);
351 extern GLfloat glutGetColor(int ndx, int component);
352 extern void glutCopyColormap(int win);
353
354 /* GLUT state retrieval sub-api */
355 extern int glutGet(GLenum type);
356 extern int glutDeviceGet(GLenum type);
357 #if (GLUT_API_VERSION >= 2)
358 /* GLUT extension support sub-API */
359 extern int glutExtensionSupported(char *name);
360 #endif
361 #if (GLUT_API_VERSION >= 3)
362 extern int glutGetModifiers(void);
363 extern int glutLayerGet(GLenum type);
364 #endif
365
366 /* GLUT font sub-API */
367 extern void glutBitmapCharacter(void *font, int character);
368 extern int glutBitmapWidth(void *font, int character);
369 extern void glutStrokeCharacter(void *font, int character);
370 extern int glutStrokeWidth(void *font, int character);
371
372 /* GLUT pre-built models sub-API */
373 extern void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
374 extern void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
375 extern void glutWireCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
376 extern void glutSolidCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
377 extern void glutWireCube(GLdouble size);
378 extern void glutSolidCube(GLdouble size);
379 extern void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius, GLint sides, GLint rings);
380 extern void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius, GLint sides, GLint rings);
381 extern void glutWireDodecahedron(void);
382 extern void glutSolidDodecahedron(void);
383 extern void glutWireTeapot(GLdouble size);
384 extern void glutSolidTeapot(GLdouble size);
385 extern void glutWireOctahedron(void);
386 extern void glutSolidOctahedron(void);
387 extern void glutWireTetrahedron(void);
388 extern void glutSolidTetrahedron(void);
389 extern void glutWireIcosahedron(void);
390 extern void glutSolidIcosahedron(void);
391
392 #ifdef __cplusplus
393 }
394
395 #endif
                            /* __glut_h__ */
396 #endif
```

54

C fglut.h FORTRAN Header File

```
1
   C Copyright (c) Mark J. Kilgard, 1994.
 2
 3
   C This program is freely distributable without licensing fees
 4
   C and is provided without guarantee or warrantee expressed or
 5 C implied. This program is -not- in the public domain.
 6
 7
   C GLUT Fortran header file
 8
 9 C display mode bit masks
10
            integer*4 GLUT_RGB
            parameter ( GLUT_RGB = 0 )
11
           integer*4 GLUT_RGBA
12
13
           parameter ( GLUT_RGBA = 0 )
14
            integer*4 GLUT_INDEX
           parameter ( GLUT_INDEX = 1 )
15
16
            integer*4 GLUT_SINGLE
17
           parameter ( GLUT\_SINGLE = 0 )
           integer*4 GLUT_DOUBLE
18
19
           parameter ( GLUT_DOUBLE = 2 )
20
            integer*4 GLUT_ACCUM
21
            parameter ( GLUT\_ACCUM = 4 )
           integer*4 GLUT_ALPHA
2.2
23
           parameter ( GLUT_ALPHA = 8 )
           integer*4 GLUT_DEPTH
24
25
           parameter ( GLUT_DEPTH = 16 )
26
            integer*4 GLUT_STENCIL
27
            parameter ( GLUT_STENCIL = 32 )
28
            integer*4 GLUT_MULTISAMPLE
29
            parameter ( GLUT_MULTISAMPLE = 128 )
30
            integer*4 GLUT_STEREO
31
            parameter ( GLUT_STEREO = 256 )
32
33 C mouse buttons
            integer*4 GLUT_LEFT_BUTTON
34
35
            parameter ( GLUT\_LEFT\_BUTTON = 0 )
36
            integer*4 GLUT_MIDDLE_BUTTON
37
            parameter ( GLUT_MIDDLE_BUTTON = 1 )
38
            integer*4 GLUT_RIGHT_BUTTON
            parameter ( GLUT_RIGHT_BUTTON = 2 )
39
40
41
   C mouse button callback state
            integer*4 GLUT_DOWN
42
43
            parameter ( GLUT_DOWN = 0 )
44
            integer*4 GLUT_UP
45
            parameter ( GLUT_UP = 1 )
46
47 C special key callback values
48
            integer*4 GLUT_KEY_F1
49
            parameter ( GLUT_KEY_F1 = 1 )
50
            integer*4 GLUT_KEY_F2
51
           parameter ( GLUT_KEY_F2 = 2 )
52
           integer*4 GLUT_KEY_F3
53
            parameter ( GLUT_KEY_F3 = 3 )
54
            integer*4 GLUT_KEY_F4
55
            parameter ( GLUT\_KEY\_F4 = 4 )
56
            integer*4 GLUT_KEY_F5
57
            parameter ( GLUT\_KEY\_F5 = 5 )
            integer*4 GLUT_KEY_F6
58
59
            parameter ( GLUT_KEY_F6 = 6 )
            integer*4 GLUT_KEY_F7
60
61
            parameter ( GLUT KEY F7 = 7 )
62
            integer*4 GLUT_KEY_F8
            parameter ( GLUT_KEY_F8 = 8 )
63
64
            integer*4 GLUT_KEY_F9
            parameter ( GLUT_KEY_F9 = 9 )
65
66
            integer*4 GLUT_KEY_F10
```

```
parameter ( GLUT_KEY_F10 = 10 )
 67
            integer*4 GLUT_KEY_F11
 68
 69
            parameter ( GLUT_KEY_F11 = 11 )
 70
            integer*4 GLUT_KEY_F12
 71
            parameter ( GLUT_KEY_F12 = 12 )
 72
            integer*4 GLUT_KEY_LEFT
 73
            parameter ( GLUT_KEY_LEFT = 100 )
            integer*4 GLUT_KEY_UP
 74
 75
            parameter ( GLUT_KEY_UP = 101 )
 76
            integer*4 GLUT_KEY_RIGHT
 77
            parameter ( GLUT_KEY_RIGHT = 102 )
 78
            integer*4 GLUT_KEY_DOWN
 79
           parameter ( GLUT_KEY_DOWN = 103 )
 80
            integer*4 GLUT_KEY_PAGE_UP
 81
            parameter ( GLUT_KEY_PAGE_UP = 104 )
 82
            integer*4 GLUT_KEY_PAGE_DOWN
 83
            parameter ( GLUT_KEY_PAGE_DOWN = 105 )
            integer*4 GLUT_KEY_HOME
 84
 85
            parameter ( GLUT_KEY_HOME = 106 )
            integer*4 GLUT_KEY_END
 86
 87
            parameter ( GLUT_KEY_END = 107 )
 88
            integer*4 GLUT_KEY_INSERT
 89
            parameter ( GLUT_KEY_INSERT = 108 )
 90
91 C entry/exit callback state
            integer*4 GLUT_LEFT
 92
93
            parameter ( GLUT\_LEFT = 0 )
 94
            integer*4 GLUT ENTERED
 95
            parameter ( GLUT_ENTERED = 1 )
96
 97 C menu usage callback state
98
            integer*4 GLUT_MENU_NOT_IN_USE
99
            parameter ( GLUT_MENU_NOT_IN_USE = 0 )
100
            integer*4 GLUT_MENU_IN_USE
101
            parameter ( GLUT_MENU_IN_USE = 1 )
102
103 C visibility callback state
104
            integer*4 GLUT_NOT_VISIBLE
105
            parameter ( GLUT_NOT_VISIBLE = 0 )
106
            integer*4 GLUT_VISIBLE
107
            parameter ( GLUT_VISIBLE = 1 )
108
109 C color index component selection values
110
            integer*4 GLUT_RED
111
            parameter ( GLUT_RED = 0 )
112
            integer*4 GLUT_GREEN
113
            parameter ( GLUT_GREEN = 1 )
114
             integer*4 GLUT_BLUE
115
            parameter ( GLUT_BLUE = 2 )
116
117 C XXX Unfortunately, SGI's Fortran compiler links with
118 C EXTERNAL data even if it is not used. This defeats
119 C the purpose of GLUT naming fonts via opaque symbols.
120 C This means GLUT Fortran programmers should explicitly
121 C declared EXTERNAL GLUT fonts in subroutines where
122 C the fonts are used.
123
124 C stroke font opaque names
125 C
            external GLUT_STROKE_ROMAN
126 C
            external GLUT_STROKE_MONO_ROMAN
127
128 C bitmap font opaque names
129 C
            external GLUT_BITMAP_9_BY_15
130 C
            external GLUT_BITMAP_8_BY_13
131 C
            external GLUT_BITMAP_TIMES_ROMAN_10
132 C
            external GLUT_BITMAP_TIMES_ROMAN_24
133 C
            external GLUT_BITMAP_HELVETICA_10
134 C
            external GLUT_BITMAP_HELVETICA_12
```

```
135 C
             external GLUT_BITMAP_HELVETICA_18
136
    C glutGet parameters
137
138
             integer*4 GLUT_WINDOW_X
139
             parameter ( GLUT_WINDOW_X = 100 )
140
             integer*4 GLUT_WINDOW_Y
141
             parameter ( GLUT_WINDOW_Y = 101 )
142
             integer*4 GLUT_WINDOW_WIDTH
143
            parameter ( GLUT_WINDOW_WIDTH = 102 )
144
             integer*4 GLUT_WINDOW_HEIGHT
145
            parameter ( GLUT_WINDOW_HEIGHT = 103 )
146
             integer*4 GLUT_WINDOW_BUFFER_SIZE
             parameter ( GLUT_WINDOW_BUFFER_SIZE = 104 )
147
             integer*4 GLUT_WINDOW_STENCIL_SIZE
148
149
             parameter ( GLUT_WINDOW_STENCIL_SIZE = 105 )
150
             integer*4 GLUT_WINDOW_DEPTH_SIZE
             parameter ( GLUT_WINDOW_DEPTH_SIZE = 106 )
151
152
             integer*4 GLUT_WINDOW_RED_SIZE
153
             parameter ( GLUT_WINDOW_RED_SIZE = 107 )
154
             integer*4 GLUT_WINDOW_GREEN_SIZE
155
            parameter ( GLUT_WINDOW_GREEN_SIZE = 108 )
156
             integer*4 GLUT_WINDOW_BLUE_SIZE
157
             parameter ( GLUT_WINDOW_BLUE_SIZE = 109 )
158
             integer*4 GLUT_WINDOW_ALPHA_SIZE
            parameter ( GLUT_WINDOW_ALPHA_SIZE = 110 )
159
160
             integer*4 GLUT_WINDOW_ACCUM_RED_SIZE
161
             parameter ( GLUT_WINDOW_ACCUM_RED_SIZE = 111 )
162
             integer*4 GLUT WINDOW ACCUM GREEN SIZE
163
             parameter ( GLUT_WINDOW_ACCUM_GREEN_SIZE = 112 )
164
             integer*4 GLUT_WINDOW_ACCUM_BLUE_SIZE
165
             parameter ( GLUT_WINDOW_ACCUM_BLUE_SIZE = 113 )
166
             integer*4 GLUT_WINDOW_ACCUM_ALPHA_SIZE
167
             parameter ( GLUT_WINDOW_ACCUM_ALPHA_SIZE = 114 )
168
             integer*4 GLUT_WINDOW_DOUBLEBUFFER
169
            parameter ( GLUT_WINDOW_DOUBLEBUFFER = 115 )
170
             integer*4 GLUT_WINDOW_RGBA
171
            parameter ( GLUT_WINDOW_RGBA = 116 )
172
             integer*4 GLUT_WINDOW_PARENT
173
             parameter ( GLUT_WINDOW_PARENT = 117 )
174
             integer*4 GLUT_WINDOW_NUM_CHILDREN
175
             parameter ( GLUT_WINDOW_NUM_CHILDREN = 118 )
176
             integer*4 GLUT_WINDOW_COLORMAP_SIZE
177
             parameter ( GLUT_WINDOW_COLORMAP_SIZE = 119 )
178
             integer*4 GLUT_WINDOW_NUM_SAMPLES
179
            parameter ( GLUT_WINDOW_NUM_SAMPLES = 120 )
180
             integer*4 GLUT_WINDOW_STEREO
181
            parameter ( GLUT_WINDOW_STEREO = 121 )
182
             integer*4 GLUT_WINDOW_CURSOR
183
             parameter ( GLUT_WINDOW_CURSOR = 122 )
184
             integer*4 GLUT_SCREEN_WIDTH
185
             parameter ( GLUT_SCREEN_WIDTH = 200 )
186
             integer*4 GLUT_SCREEN_HEIGHT
187
             parameter ( GLUT_SCREEN_HEIGHT = 201 )
188
             integer*4 GLUT_SCREEN_WIDTH_MM
189
            parameter ( GLUT_SCREEN_WIDTH_MM = 202 )
190
             integer*4 GLUT_SCREEN_HEIGHT_MM
191
             parameter ( GLUT SCREEN HEIGHT MM = 203 )
192
             integer*4 GLUT_MENU_NUM_ITEMS
193
             parameter ( GLUT_MENU_NUM_ITEMS = 300 )
194
             integer*4 GLUT_DISPLAY_MODE_POSSIBLE
195
             parameter ( GLUT_DISPLAY_MODE_POSSIBLE = 400 )
196
             integer*4 GLUT_INIT_WINDOW_X
197
             parameter ( GLUT_INIT_WINDOW_X = 500 )
198
             integer*4 GLUT_INIT_WINDOW_Y
199
             parameter ( GLUT_INIT_WINDOW_Y = 501 )
200
             integer*4 GLUT_INIT_WINDOW_WIDTH
201
             parameter ( GLUT_INIT_WINDOW_WIDTH = 502 )
```

integer*4 GLUT_INIT_WINDOW_HEIGHT

202

```
203
            parameter ( GLUT_INIT_WINDOW_HEIGHT = 503 )
204
            integer*4 GLUT_INIT_DISPLAY_MODE
205
            parameter ( GLUT_INIT_DISPLAY_MODE = 504 )
206
            integer*4 GLUT_ELAPSED_TIME
207
            parameter ( GLUT_ELAPSED_TIME = 700 )
208
209 C glutDeviceGet parameters
210
            integer*4 GLUT_HAS_KEYBOARD
211
            parameter ( GLUT_HAS_KEYBOARD = 600 )
212
            integer*4 GLUT_HAS_MOUSE
213
            parameter ( GLUT_HAS_MOUSE = 601 )
214
            integer*4 GLUT_HAS_SPACEBALL
215
            parameter ( GLUT_HAS_SPACEBALL = 602 )
216
            integer*4 GLUT_HAS_DIAL_AND_BUTTON_BOX
217
            parameter ( GLUT_HAS_DIAL_AND_BUTTON_BOX = 603 )
218
            integer*4 GLUT_HAS_TABLET
            parameter ( GLUT_HAS_TABLET = 604 )
219
220
            integer*4 GLUT_NUM_MOUSE_BUTTONS
221
            parameter ( GLUT_NUM_MOUSE_BUTTONS = 605 )
            integer*4 GLUT_NUM_SPACEBALL_BUTTONS
222
223
            parameter ( GLUT_NUM_SPACEBALL_BUTTONS = 606 )
224
            integer*4 GLUT_NUM_BUTTON_BOX_BUTTONS
225
            parameter ( GLUT_NUM_BUTTON_BOX_BUTTONS = 607 )
226
            integer*4 GLUT_NUM_DIALS
227
            parameter ( GLUT_NUM_DIALS = 608 )
228
            integer*4 GLUT_NUM_TABLET_BUTTONS
229
            parameter ( GLUT_NUM_TABLET_BUTTONS = 609 )
230
231 C glutLayerGet parameters
            integer*4 GLUT_OVERLAY_POSSIBLE
232
233
            parameter ( GLUT_OVERLAY_POSSIBLE = 800 )
234
            integer*4 GLUT_LAYER_IN_USE
235
            parameter ( GLUT_LAYER_IN_USE = 801 )
236
            integer*4 GLUT_HAS_OVERLAY
            parameter ( GLUT_HAS_OVERLAY = 802 )
237
238
            integer*4 GLUT_TRANSPARENT_INDEX
239
            parameter ( GLUT_TRANSPARENT_INDEX = 803 )
240
            integer*4 GLUT_NORMAL_DAMAGED
241
            parameter ( GLUT_NORMAL_DAMAGED = 804 )
242
            integer*4 GLUT_OVERLAY_DAMAGED
243
            parameter ( GLUT_OVERLAY_DAMAGED = 805 )
244
245 C glutUseLayer parameters
246
            integer*4 GLUT_NORMAL
247
            parameter ( GLUT_NORMAL = 0 )
248
            integer*4 GLUT_OVERLAY
249
            parameter ( GLUT_OVERLAY = 1 )
250
251 C glutGetModifiers return mask
252
            integer*4 GLUT_ACTIVE_SHIFT
253
            parameter ( GLUT_ACTIVE_SHIFT = 1 )
254
            integer*4 GLUT_ACTIVE_CTRL
255
            parameter ( GLUT_ACTIVE_CTRL = 2 )
            integer*4 GLUT_ACTIVE_ALT
256
257
            parameter ( GLUT_ACTIVE_ALT = 4 )
258
259 C glutSetCursor parameters
260
            integer*4 GLUT_CURSOR_RIGHT_ARROW
            parameter ( GLUT_CURSOR_RIGHT_ARROW = 0 )
261
262
            integer*4 GLUT_CURSOR_LEFT_ARROW
263
            parameter ( GLUT_CURSOR_LEFT_ARROW = 1 )
264
            integer*4 GLUT_CURSOR_INFO
265
            parameter ( GLUT_CURSOR_INFO = 2 )
266
            integer*4 GLUT_CURSOR_DESTROY
267
            parameter ( GLUT_CURSOR_DESTROY = 3 )
268
            integer*4 GLUT_CURSOR_HELP
269
            parameter ( GLUT_CURSOR_HELP = 4 )
            integer*4 GLUT_CURSOR_CYCLE
270
```

```
271
            parameter ( GLUT_CURSOR_CYCLE = 5 )
272
            integer*4 GLUT_CURSOR_SPRAY
            parameter ( GLUT\_CURSOR\_SPRAY = 6 )
273
274
            integer*4 GLUT_CURSOR_WAIT
275
             parameter ( GLUT_CURSOR_WAIT = 7 )
276
            integer*4 GLUT_CURSOR_TEXT
277
            parameter ( GLUT_CURSOR_TEXT = 8 )
278
             integer*4 GLUT_CURSOR_CROSSHAIR
279
            parameter ( GLUT_CURSOR_CROSSHAIR = 9 )
280
             integer*4 GLUT_CURSOR_UP_DOWN
281
            parameter ( GLUT_CURSOR_UP_DOWN = 10 )
282
            integer*4 GLUT_CURSOR_LEFT_RIGHT
283
            parameter ( GLUT_CURSOR_LEFT_RIGHT = 11 )
284
             integer*4 GLUT_CURSOR_TOP_SIDE
285
             parameter ( GLUT_CURSOR_TOP_SIDE = 12 )
286
             integer*4 GLUT_CURSOR_BOTTOM_SIDE
287
            parameter ( GLUT_CURSOR_BOTTOM_SIDE = 13 )
288
             integer*4 GLUT_CURSOR_LEFT_SIDE
289
            parameter ( GLUT_CURSOR_LEFT_SIDE = 14 )
290
             integer*4 GLUT_CURSOR_RIGHT_SIDE
291
            parameter ( GLUT_CURSOR_RIGHT_SIDE = 15 )
292
             integer*4 GLUT_CURSOR_TOP_LEFT_CORNER
293
            parameter ( GLUT_CURSOR_TOP_LEFT_CORNER = 16 )
294
             integer*4 GLUT_CURSOR_TOP_RIGHT_CORNER
295
            parameter ( GLUT_CURSOR_TOP_RIGHT_CORNER = 17 )
296
             integer*4 GLUT_CURSOR_BOTTOM_RIGHT_CORNER
297
            parameter ( GLUT_CURSOR_BOTTOM_RIGHT_CORNER = 18 )
298
             integer*4 GLUT_CURSOR_BOTTOM_LEFT_CORNER
299
             parameter ( GLUT_CURSOR_BOTTOM_LEFT_CORNER = 19 )
300
            integer*4 GLUT_CURSOR_INHERIT
301
            parameter ( GLUT_CURSOR_INHERIT = 100 )
302
             integer*4 GLUT_CURSOR_NONE
303
            parameter ( GLUT_CURSOR_NONE = 101 )
304
             integer*4 GLUT_CURSOR_FULL_CROSSHAIR
            parameter ( GLUT_CURSOR_FULL_CROSSHAIR = 102 )
305
306
307 C GLUT functions
308
             integer*4 glutcreatewindow
309
             integer*4 glutgetwindow
             integer*4 glutcreatemenu
310
311
             integer*4 glutgetmenu
312
            real glutgetcolor
313
             integer*4 glutget
314
             integer*4 glutdeviceget
315
            integer*4 glutextensionsupported
316
317 C GLUT NULL name
318
            external glutnull
319
```

References

- [1] Kurt Akeley, "RealityEngine Graphics," Proceedings of SIGGRAPH '93, July 1993.
- [2] Edward Angel, Interactive Computer Graphics: A top-down approach with OpenGL, Addison-Wesley, ISBN 0-201-85571-2, 1996.
- [3] F.C. Crow, "The Origins of the Teapot," IEEE Computer Graphics and Applications, January 1987.
- [4] Phil Karlton, OpenGL Graphics with the X Window System, Ver. 1.0, Silicon Graphics, April 30, 1993.
- [5] Mark J. Kilgard, "Going Beyond the MIT Sample Server: The Silicon Graphics X11 Server," *The X Journal*, SIGS Publications, January 1993.
- [6] Mark Kilgard, "Programming X Overlay Windows," The X Journal, SIGS Publications, July 1993.
- [7] Mark Kilgard, "OpenGL and X, Part 2: Using OpenGL with Xlib," *The X Journal*, SIGS Publications, Jan/Feb 1994.
- [8] Mark Kilgard, "OpenGL and X, Part 3: Integrating OpenGL with Motif," *The X Journal*, SIGS Publications, Jul/Aug 1994.
- [9] Mark Kilgard, "An OpenGL Toolkit," The X Journal, SIGS Publications, Nov/Dec 1994.
- [10] Mark Kilgard, Programming OpenGL for the X Window System, Addison-Wesley, ISBN 0-201-48359-9, 1996.
- [11] Jackie Neider, Tom Davis, Mason Woo, *OpenGL Programming Guide: The official guide to learning OpenGL, Release 1*, Addison Wesley, 1993.
- [12] OpenGL Architecture Review Board, OpenGL Reference Manual: The official reference document for OpenGL, Release 1, Addison Wesley, 1992.
- [13] Mark Patrick, George Sachs, X11 Input Extension Library Specification, X Consortium Standard, X11R6, April 18, 1994.
- [14] Mark Patrick, George Sachs, X11 Input Extension Protocol Specification, X Consortium Standard, X11R6, April 17, 1994.
- [15] Robert Scheifler, James Gettys, X Window System: The complete Reference to Xlib, X Protocol, ICCCM, XLFD, third edition, Digital Press, 1992.
- [16] Mark Segal, Kurt Akeley, The OpenGLTM Graphics System: A Specification, Version 1.0, Silicon Graphics, June 30, 1992.
- [17] Silicon Graphics, Graphics Library Programming Guide, Document Number 007-1210-040, 1991.
- [18] Silicon Graphics, Graphics Library Window and Font Library Guide, Document Number 007-1329-010, 1991.

Index

_MOTIF_WM_HINTS, 12 _SGI_CROSSHAIR_CURSOR, 14 __glutFatalError, 40 __glutWarning, 40 Architectural Review Board, 41 Callback, 4 Colormap, 4 Dials and button box, 4 Display mode, 4 glFlush, 11, 40 GLUT_LUMINANCE, 3, 8 glutAddMenuEntry, 17 glutAddSubMenu, 18 glutAttachMenu, 19 glutBitmapCharacter, 34 glutBitmapWidth, 3, 35 glutButtonBoxFunc, 26 glutChangeToMenuEntry, 18 glutChangeToSubMenu, 18 glutCopyColormap, 30 glutCreateMenu, 16 glutCreateSubWindow, 9, 40 glutCreateWindow, 9, 40 glutDestroyMenu, 17 glutDestroyWindow, 10 glutDeviceGet, 32, 44 glutDialsFunc, 26 glutDisplayFunc, 4, 20 glutEntryFunc, 23 glutEstablishOverlay, 3, 14 glutExtensionSupported, 33 glutFullScreen, 12 glutGet, 30, 40, 44 glutGetColor, 29 glutGetMenu, 17, 40 glutGetModifiers, 3, 33 glutGetWindow, 10, 40 glutHideOverlay, 3, 16 glutHideWindow, 13 glutIconifyWindow, 13 glutIdleFunc, 28 glutInit, 6 glutInitDisplayMode, 3, 7 glutInitWindowPosition,7 glutInitWindowSize, 6, 7 glutKeyboardFunc, 21, 40 glutLayerGet, 3, 32, 44 glutMainLoop, 8

glutMenuStateFunc, 3, 40 glutMenuStatusFunc, 3, 27 glutMotionFunc, 22 glutMouseFunc, 22 GLUTNULL, 42 glutOverlayDisplayFunc, 20 glutPopWindow, 12 glutPositionWindow, 11 glutPostOverlayRedisplay, 3, 16 glutPostRedisplay, 10, 39 glutPushWindow, 12 glutRemoveMenuItem, 19 glutRemoveOverlay, 3, 15 glutReshapeFunc, 21 glutReshapeWindow, 11 glutSetColor, 29 glutSetCursor, 13, 14 glutSetIconTitle, 13 glutSetMenu, 17, 40 glutSetWindow, 10, 40 glutSetWindowTitle, 13 glutShowOverlay, 3, 16 glutShowWindow, 13 glutSolidCone, 37 glutSolidCube, 36 glutSolidDodecahedron, 38 glutSolidIcosahedron, 38 glutSolidOctahedron, 38 glutSolidSphere, 36 glutSolidTeapot, 39 glutSolidTetrahedron, 38 glutSolidTorus, 37 glutSpaceballButtonFunc, 25 glutSpaceballMotionFunc, 24 glutSpaceballRotateFunc, 25 glutSpecialFunc, 24, 40 glutStrokeBitmap, 3 glutStrokeCharacter, 35 glutStrokeWidth, 36 glutSwapBuffers, 11, 40 glutTabletButtonFunc, 27 glutTabletMotionFunc, 27 glutTimerFunc, 28 glutUseLayer, 15 glutUseOverlay, 3 glutVisibilityFunc, 23 glutWireCone, 37 glutWireCube, 36 glutWireDodecahedron, 38 glutWireIcosahedron, 38 glutWireOctahedron, 38

glutWireSphere, 36 glutWireTeapot, 39 glutWireTetrahedron, 38 glutWireTorus, 37 Idle, 4 Layer in use, 4 Menu entry, 4 Menu item, 4 Modifiers, 5 Multisampling, 5 Normal plane, 5 onexit, 40 OpenGL errors, 7 Overlay, 5 overlay hardware, 14 Pop, 5 Pop-up menu, 5 Push, 5 Reshape, 5 SERVER_OVERLAY_VISUALS, 15, 17 Spaceball, 5 Stereo, 5 Sub-menu, 5 Sub-menu trigger, 5 Subwindow, 5 Tablet, 5 The X Journal, 1 Timer, 5 Top-level window, 5 Window, 5 Window display state, 5 Window system, 5 WM_COMMAND, 9 X Input Extension, 20 X Inter-Client Communication Conventions Manual, 9

X protocol errors, 7

62

$\begin{array}{c} {\rm OpenGL}^{\circledast} \ {\rm Graphics \ with \ the \ X \ Window \ System}^{\circledast} \\ {\rm (Version \ 1.3)} \end{array}$

Document Editors (version 1.3): Paula Womack, Jon Leech

Copyright © 1992-1998 Silicon Graphics, Inc.

This document contains unpublished information of Silicon Graphics, Inc.

This document is protected by copyright, and contains information proprietary to Silicon Graphics, Inc. Any copying, adaptation, distribution, public performance, or public display of this document without the express written consent of Silicon Graphics, Inc. is strictly prohibited. The receipt or possession of this document does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

U.S. Government Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions set forth in FAR 52.227.19(c)(2) or subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227.7013 and/or in similar or successor clauses in the FAR or the DOD or NASA FAR Supplement. Unpublished rights reserved under the copyright laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94039-7311.

OpenGL is a registered trademark of Silicon Graphics, Inc. Unix is a registered trademark of The Open Group. The "X" device and X Windows System are trademarks of The Open Group.

Contents

1	Ove	rview		1
2	GLX	C Oper	ation	2
	2.1	Render	ring Contexts and Drawing Surfaces	2
	2.2	Using 1	Rendering Contexts	3
	2.3	Direct	Rendering and Address Spaces	4
	2.4	OpenG	L Display Lists	5
	2.5		e Objects	6
	2.6	Alignir	ng Multiple Drawables	7
	2.7	Multip	le Threads	7
3	Fune	ctions	and Errors	9
	3.1	Errors		9
	3.2	Events		10
	3.3	Functio	ons	10
		3.3.1	Initialization	10
		3.3.2	GLX Versioning	11
		3.3.3	Configuration Management	12
		3.3.4	On Screen Rendering	21
		3.3.5		21
		3.3.6		25
		3.3.7	Rendering Contexts	25
		3.3.8	Events	31
		3.3.9	Synchronization Primitives	33
		3.3.10	Double Buffering	33
		3.3.11	Access to X Fonts	34
	3.4	Backwa	ards Compatibility	35
		3.4.1		35
		3.4.2	Off Screen Rendering	39

CONTENTS

	3.5	Rendering Contexts	40
4	Enc	coding on the X Byte Stream	42
	4.1	Requests that hold a single extension request	42
	4.2	Request that holds multiple OpenGL commands	43
	4.3	Wire representations and byte swapping	44
	4.4	Sequentiality	44
5	\mathbf{Ext}	ending OpenGL	47
6	GL	X Versions	49
	6.1	New Commands in GLX Version 1.1	49
	6.2	New Commands in GLX Version 1.2	49
	6.3	New Commands in GLX Version 1.3	50
7	Glo	ssary	51
	Ind	ex of GLX Commands	53

ii

List of Figures

2.1	Direct and Indirect Rendering Block Diagram.	4
4.1	GLX byte stream.	43

List of Tables

3.1	GLXFBConfig attributes	13
3.2	Types of Drawables Supported by GLXFBConfig	14
3.3	Mapping of Visual Types to GLX tokens.	14
3.4	Default values and match criteria for GLXFBConfig attributes.	19
3.5	Context attributes	30
3.6	Masks identifying clobbered buffers.	32
3.7	GLX attributes for Visuals	36
3.8	Defaults and selection criteria used by $\mathbf{glXChooseVisual}$.	38
6.1	Relationship of OpenGL and GLX versions	49

Chapter 1

Overview

This document describes GLX, the OpenGL extension to the X Window System. It refers to concepts discussed in the OpenGL specification, and may be viewed as an X specific appendix to that document. Parts of the document assume some acquaintance with both OpenGL and X.

In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense: connection and authentication are accomplished with the normal X mechanisms. As with other X extensions, there is a defined network protocol for the OpenGL rendering commands encapsulated within the X byte stream.

Since performance is critical in 3D rendering, there is a way for OpenGL rendering to bypass the data encoding step, the data copying, and interpretation of that data by the X server. This *direct rendering* is possible only when a process has direct access to the graphics pipeline. Allowing for parallel rendering has affected the design of the GLX interface. This has resulted in an added burden on the client to explicitly prevent parallel execution when such execution is inappropriate.

X and OpenGL have different conventions for naming entry points and macros. The GLX extension adopts those of OpenGL.

Chapter 2

GLX Operation

2.1 Rendering Contexts and Drawing Surfaces

The OpenGL specification is intentionally vague on how a *rendering context* (an abstract OpenGL state machine) is created. One of the purposes of GLX is to provide a means to create an OpenGL context and associate it with a drawing surface.

In X, a rendering surface is called a Drawable. X provides two types of Drawables: Windows which are located onscreen and Pixmaps which are maintained offscreen. The GLX equivalent to a Window is a GLXWindow and the GLX equivalent to a Pixmap is a GLXPixmap. GLX introduces a third type of drawable, called a GLXPbuffer, for which there is no X equivalent. GLXPbuffers are used for offscreen rendering but they have different semantics than GLXPixmaps that make it easier to allocate them in non-visible frame buffer memory.

GLXWindows, GLXPixmaps and GLXPbuffers are created with respect to a GLXFBConfig; the GLXFBConfig describes the depth of the color buffer components and the types, quantities and sizes of the *ancillary buffers* (i.e., the depth, accumulation, auxiliary, and stencil buffers). Double buffering and stereo capability is also fixed by the GLXFBConfig.

Ancillary buffers are associated with a GLXDrawable, not with a rendering context. If several rendering contexts are all writing to the same window, they will share those buffers. Rendering operations to one window never affect the unobscured pixels of another window, or the corresponding pixels of ancillary buffers of that window. If an Expose event is received by the client, the values in the ancillary buffers and in the back buffers for regions corresponding to the exposed region become undefined. A rendering context can be used with any GLXDrawable that it is *compatible* with (subject to the restrictions discussed in the section on address space and the restrictions discussed under glXCreatePixmap). A drawable and context are compatible if they

- support the same type of rendering (e.g., RGBA or color index)
- have color buffers and ancillary buffers of the same depth. For example, a GLXDrawable that has a front left buffer and a back left buffer with red, green and blue sizes of 4 would not be compatible with a context that was created with a visual or GLXFBConfig that has only a front left buffer with red, green and blue sizes of 8. However, it would be compatible with a context that was created with a GLXFBConfig that has only a front left buffer if the red, green and blue sizes are 4.
- were created with respect to the same X screen

As long as the compatibility constraint is satisfied (and the address space requirement is satisfied), applications can render into the same GLXDrawable, using different rendering contexts. It is also possible to use a single context to render into multiple GLXDrawables.

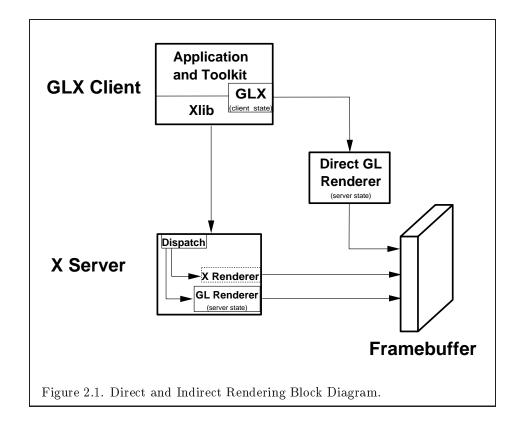
For backwards compatibility with GLX versions 1.2 and earlier, a rendering context can also be used to render into a Window. Thus, a GLXDrawable is the union {GLXWindow, GLXPixmap, GLXPbuffer, Window}. In X, Windows are associated with a Visual. In GLX the definition of Visual has been extended to include the types, quantities and sizes of the ancillary buffers and information indicating whether or not the Visual is double buffered. For backwards compatibility, a GLXPixmap can also be created using a Visual.

2.2 Using Rendering Contexts

OpenGL defines both client state and server state. Thus a rendering context consists of two parts: one to hold the client state and one to hold the server state.

Each thread can have at most one current rendering context. In addition, a rendering context can be current for only one thread at a time. The client is responsible for creating a rendering context and a drawable.

Issuing OpenGL commands may cause the X buffer to be flushed. In particular, calling **glFlush** when indirect rendering is occurring, will flush both the X and OpenGL rendering streams.



Some state is shared between the OpenGL and X. The pixel values in the X frame buffer are shared. The X double buffer extension (DBE) has a definition for which buffer is currently the displayed buffer. This information is shared with GLX. The state of which buffer is displayed tracks in both extensions, independent of which extension initiates a buffer swap.

2.3 Direct Rendering and Address Spaces

One of the basic assumptions of the X protocol is that if a client can name an object, then it can manipulate that object. GLX introduces the notion of an *Address Space*. A GLX object cannot be used outside of the address space in which it exists.

In a classic UNIX environment, each process is in its own address space. In a multi-threaded environment, each of the threads will share a virtual address space which references a common data region. An OpenGL client that is rendering to a graphics engine directly connected to the executing CPU may avoid passing the tokens through the X server. This generalization is made for performance reasons. The model described here specifically allows for such optimizations, but does not mandate that any implementation support it.

When direct rendering is occurring, the address space of the OpenGL implementation is that of the direct process; when direct rendering is not being used (i.e., when indirect rendering is occurring), the address space of the OpenGL implementation is that of the X server. The client has the ability to reject the use of direct rendering, but there may be a performance penalty in doing so.

In order to use direct rendering, a client must create a direct rendering context (see figure 2.1). Both the client context state and the server context state of a direct rendering context exist in the client's address space; this state cannot be shared by a client in another process. With indirect rendering contexts, the client context state is kept in the client's address space and the server context state is kept in the address space of the X server. In this case the server context state is stored in an X resource; it has an associated XID and may potentially be used by another client process.

Although direct rendering support is optional, all implementations are required to support indirect rendering.

2.4 OpenGL Display Lists

Most OpenGL state is small and easily retrieved using the **glGet*** commands. This is not true of OpenGL display lists, which are used, for example, to encapsulate a model of some physical object. First, there is no mechanism to obtain the contents of a display list from the rendering context. Second, display lists may be large and numerous. It may be desirable for multiple rendering contexts to share display lists rather than replicating that information in each context.

GLX provides for limited sharing of display lists. Since the lists are part of the server context state they can be shared only if the server state for the sharing contexts exists in a single address space. Using this mechanism, a single set of lists can be used, for instance, by a context that supports color index rendering and a context that supports RGBA rendering.

When display lists are shared between OpenGL contexts, the sharing extends only to the display lists themselves and the information about which display list numbers have been allocated. In particular, the value of the base set with **glListBase** is not shared.

Note that the list named in a **glNewList** call is not created or superseded until **glEndList** is called. Thus if one rendering context is sharing a display list with another, it will continue to use the existing definition while the second context is in the process of re-defining it. If one context deletes a list that is being executed by another context, the second context will continue executing the old contents of the list until it reaches the end.

A group of shared display lists exists until the last referencing rendering context is destroyed. All rendering contexts have equal access to using lists or defining new lists. Implementations sharing display lists must handle the case where one rendering context is using a display list when another rendering context destroys that list or redefines it.

In general, OpenGL commands are not guaranteed to be atomic. The operation of **glEndList** and **glDeleteLists** are exceptions: modifications to the shared context state as a result of executing **glEndList** or **glDeleteLists** are atomic.

2.5 Texture Objects

OpenGL texture state can be encapsulated in a named texture object. A texture object is created by binding an unused name to one of the texture targets (GL_TEXTURE_1D, GL_TEXTURE_2D or GL_TEXTURE_3D) of a rendering context. When a texture object is bound, OpenGL operations on the target to which it is bound affect the bound texture object, and queries of the target to which it is bound return state from the bound texture object.

Texture objects may be shared by rendering contexts, as long as the server portion of the contexts share the same address space. (Like display lists, texture objects are part of the server context state.) OpenGL makes no attempt to synchronize access to texture objects. If a texture object is bound to more than one context, then it is up to the programmer to ensure that the contents of the object are not being changed via one context while another context is using the texture object for rendering. The results of changing a texture object while another context is using it are undefined.

All modifications to shared context state as a result of executing **glBind**-**Texture** are atomic. Also, a texture object will not be deleted until it is no longer bound to any rendering context.

6

2.6 Aligning Multiple Drawables

A client can create one window in the overlay planes and a second in the main planes and then move them independently or in concert to keep them aligned. To keep the overlay and main plane windows aligned, the client can use the following paradigm:

- Make the windows which are to share the same screen area children of a single window (that will never be written). Size and position the children to completely occlude their parent. When the window combination must be moved or resized, perform the operation on the parent.
- Make the subwindows have a background of None so that the X server will not paint into the shared area when you restack the children.
- Select for device-related events on the parent window, not on the children. Since device-related events with the focus in one of the child windows will be inherited by the parent, input dispatching can be done directly without reference to the child on top.

2.7 Multiple Threads

It is possible to create a version of the client side library that is protected against multiple threads attempting to access the same connection. This is accomplished by having appropriate definitions for **LockDisplay** and **UnlockDisplay**. Since there is some performance penalty for doing the locking, it is implementation-dependent whether a thread safe version, a non-safe version, or both versions of the library are provided. Interrupt routines may not share a connection (and hence a rendering context) with the main thread. An application may be written as a set of co-operating processes.

X has atomicity (between clients) and sequentiality (within a single client) requirements that limit the amount of parallelism achievable when interpreting the command streams. GLX relaxes these requirements. Sequentiality is still guaranteed within a command stream, but not between the X and the OpenGL command streams. It is possible, for example, that an X command issued by a single threaded client after an OpenGL command might be executed before that OpenGL command.

The X specification requires that commands are atomic:

If a server is implemented with internal concurrency, the overall effect must be as if individual requests are executed to completion in some serial order, and requests from a given connection must be executed in delivery order (that is, the total execution order is a shuffle of the individual streams).

OpenGL commands are not guaranteed to be atomic. Some OpenGL rendering commands might otherwise impair interactive use of the windowing system by the user. For instance calling a deeply nested display list or rendering a large texture mapped polygon on a system with no graphics hardware could prevent a user from popping up a menu soon enough to be usable.

Synchronization is in the hands of the client. It can be maintained with moderate cost with the judicious use of the glFinish, glXWaitGL, glXWaitX, and XSync commands. OpenGL and X rendering can be done in parallel as long as the client does not preclude it with explicit synchronization calls. This is true even when the rendering is being done by the X server. Thus, a multi-threaded X server implementation may execute OpenGL rendering commands in parallel with other X requests.

Some performance degradation may be experienced if needless switching between OpenGL and X rendering is done. This may involve a round trip to the server, which can be costly.

Chapter 3

Functions and Errors

3.1 Errors

Where possible, as in X, when a request terminates with an error, the request has no side effects.

The error codes that may be generated by a request are described with that request. The following table summarizes the GLX-specific error codes that are visible to applications:

GLXBadContext A value for a Context argument does not name a Context.

- GLXBadContextState An attempt was made to switch to another rendering context while the current context was in glRenderMode GL_FEEDBACK or GL_SELECT, or a call to glXMakeCurrent was made between a glBegin and the corresponding call to glEnd.
- GLXBadCurrentDrawable The current Drawable of the calling thread is a window or pixmap that is no longer valid.
- GLXBadCurrentWindow The current Window of the calling thread is a window that is no longer valid. This error is being deprecated in favor of GLXBadCurrentDrawable.
- GLXBadDrawable The Drawable argument does not name a Drawable configured for OpenGL rendering.
- GLXBadFBConfig The GLXFBConfig argument does not name a GLXFBConfig.
- GLXBadPbuffer The GLXPbuffer argument does not name a GLXPbuffer.

- GLXBadPixmap The Pixmap argument does not name a Pixmap that is appropriate for OpenGL rendering.
- GLXUnsupportedPrivateRequest May be returned in response to either a glXVendorPrivate request or a glXVendorPrivateWithReply request.
- GLXBadWindow The GLXWindow argument does not name a GLXWindow.

The following error codes may be generated by a faulty GLX implementation, but would not normally be visible to clients:

- GLXBadContextTag A rendering request contains an invalid context tag. (Context tags are used to identify contexts in the protocol.)
- GLXBadRenderRequest A glXRender request is ill-formed.

GLXBadLargeRequest A glXRenderLarge request is ill-formed.

3.2 Events

GLX introduces one new event:

GLX_PbufferClobber The given pbuffer has been removed from framebuffer memory and may no longer be valid. These events are generated as a result of conflicts in the framebuffer allocation between two drawables when one or both of the drawables are pbuffers.

3.3 Functions

GLX functions should not be called between **glBegin** and **glEnd** operations. If a GLX function is called within a **glBegin/glEnd** pair, then the result is undefined; however, no error is reported.

3.3.1 Initialization

To ascertain if the GLX extension is defined for an X server, use

dpy specifies the connection to the X server. False is returned if the extension is not present. *error_base* is used to return the value of the first error code and *event_base* is used to return the value of the first event code. The constant error codes and event codes should be added to these base values to get the actual value.

The GLX definition exists in multiple versions. Use

Bool glXQueryVersion(Display *dpy, int *major, int *minor);

to discover which version of GLX is available. Upon success, major and minor are filled in with the major and minor versions of the extension implementation. If the client and server both have the same major version number then they are compatible and the minor version that is returned is the minimum of the two minor version numbers.

major and minor do not return values if they are specified as NULL.

glXQueryVersion returns True if it succeeds and False if it fails. If it fails, *major* and *minor* are not updated.

3.3.2 GLX Versioning

The following functions are available only if the GLX version is 1.1 or later:

glXQueryExtensionsString returns a pointer to a string describing which GLX extensions are supported on the connection. The string is zero-terminated and contains a space-seperated list of extension names. The extension names themselves do not contain spaces. If there are no extensions to GLX, then the empty string is returned.

```
const char *glXGetClientString(Display *dpy, int
    name);
```

glXGetClientString returns a pointer to a static, zero-terminated string describing some aspect of the client library. The possible values for *name* are GLX_VENDOR, GLX_VERSION, and GLX_EXTENSIONS. If *name* is not set to one of these values then NULL is returned. The format and contents of the vendor string is implementation dependent, and the format of the extension string is the same as for glXQueryExtensionsString. The version string is laid out as follows:

<major_version.minor_version><space><vendor-specific info>

Both the major and minor portions of the version number are of arbitrary length. The vendor-specific information is optional. However, if it is present, the format and contents are implementation specific.

const char* glXQueryServerString(Display *dpy, int screen, int name);

glXQueryServerString returns a pointer to a static, zero-terminated string describing some aspect of the server's GLX extension. The possible values for *name* and the format of the strings is the same as for glXGet-ClientString. If *name* is not set to a recognized value then NULL is returned.

3.3.3 Configuration Management

A GLXFBConfig describes the format, type and size of the color buffers and ancillary buffers for a GLXDrawable. When the GLXDrawable is a GLXWindow then the GLXFBConfig that describes it has an associated X Visual; for GLXPixmaps and GLXPbuffers there may or may not be an X Visual associated with the GLXFBConfig.

The attributes for a GLXFBConfig are shown in Table 3.1. The constants shown here are passed to glXGetFBConfigs and glXChooseFBConfig to specify which attributes are being queried.

GLX_BUFFER_SIZE gives the total depth of the color buffer in bits. For GLXFBConfigs that correspond to a PseudoColor or StaticColor visual, this is equal to the depth value reported in the core X11 Visual. For GLXFBConfigs that correspond to a TrueColor or DirectColor visual, GLX_BUFFER_SIZE is the sum of GLX_RED_SIZE, GLX_GREEN_SIZE, GLX_BLUE_SIZE, and GLX_ALPHA_SIZE. Note that this value may be larger than the depth value reported in the core X11 visual since it may include alpha planes that may not be reported by X11. Also, for GLXFBConfigs that correspond to a TrueColor visual, the sum of GLX_RED_SIZE, GLX_GREEN_SIZE, and GLX_BLUE_SIZE may be larger than the maximum depth that core X11 can support.

The attribute GLX_RENDER_TYPE has as its value a mask indicating what type of GLXContext a drawable created with the corresponding GLXFBConfig can be bound to. The following bit settings are supported: GLX_RGBA_BIT and GLX_COLOR_INDEX_BIT. If both of these bits are set in the mask then drawables created with the GLXFBConfig can be bound to both RGBA and color index rendering contexts.

Attribute	Type	Notes	
GLX_FBCONFIG_ID	XID	XID of GLXFBConfig	
GLX_BUFFER_SIZE	integer	depth of the color buffer	
GLX_LEVEL	integer	frame buffer level	
GLX_DOUBLEBUFFER	boolean	True if color buffers	
		have front/back pairs	
GLX_STERE0	boolean	True if color buffers	
		have left/right pairs	
GLX_AUX_BUFFERS	integer	no. of auxiliary color buffers	
GLX_RED_SIZE	integer	no. of bits of Red in the color buffer	
GLX_GREEN_SIZE	integer	no. of bits of Green in the color buffer	
GLX_BLUE_SIZE	integer	no. of bits of Blue in the color buffer	
GLX_ALPHA_SIZE	integer	no. of bits of Alpha in the color buffer	
GLX_DEPTH_SIZE	integer	no. of bits in the depth buffer	
GLX_STENCIL_SIZE	integer	no. of bits in the stencil buffer	
GLX_ACCUM_RED_SIZE	integer	no. Red bits in the accum. buffer	
GLX_ACCUM_GREEN_SIZE	integer	no. Green bits in the accum. buffer	
GLX_ACCUM_BLUE_SIZE	integer	no. Blue bits in the accum. buffer	
GLX_ACCUM_ALPHA_SIZE	integer	no. of Alpha bits in the accum. buffer	
GLX_RENDER_TYPE	bitmask	which rendering modes are supported.	
GLX_DRAWABLE_TYPE	bitmask	which GLX drawables are supported.	
GLX_X_RENDERABLE	boolean	True if X can render to drawable	
GLX_X_VISUAL_TYPE	integer	X visual type of the associated visual	
GLX_CONFIG_CAVEAT	enum	any caveats for the configuration	
GLX_TRANSPARENT_TYPE	enum	type of transparency supported	
GLX_TRANSPARENT_INDEX_VALUE	integer	transparent index value	
GLX_TRANSPARENT_RED_VALUE	integer	transparent red value	
GLX_TRANSPARENT_GREEN_VALUE	integer	transparent green value	
GLX_TRANSPARENT_BLUE_VALUE	integer	transparent blue value	
GLX_TRANSPARENT_ALPHA_VALUE	integer	transparent alpha value	
GLX_MAX_PBUFFER_WIDTH	integer	maximum width of GLXPbuffer	
GLX_MAX_PBUFFER_HEIGHT	integer	maximum height of GLXPbuffer	
GLX_MAX_PBUFFER_PIXELS	integer	maximum size of GLXPbuffer	
GLX_VISUAL_ID	integer	XID of corresponding Visual	

Table 3.1: ${\tt GLXFBConfig}$ attributes.

GLX Token Name	Description	
GLX_WINDOW_BIT	GLXFBConfig supports windows	
GLX_PIXMAP_BIT	GLXFBConfig supports pixmaps	
GLX_PBUFFER_BIT	GLXFBConfig supports pbuffers	

Table 3.2: Types of Drawables Supported by GLXFBConfig

GLX Token Name	X Visual Type	
GLX_TRUE_COLOR	TrueColor	
GLX_DIRECT_COLOR	DirectColor	
GLX_PSEUDO_COLOR	PseudoColor	
GLX_STATIC_COLOR	StaticColor	
GLX_GRAY_SCALE	GrayScale	
GLX_STATIC_GRAY	StaticGray	
GLX_X_VISUAL_TYPE	No associated Visual	

Table 3.3: Mapping of Visual Types to GLX tokens.

The attribute GLX_DRAWABLE_TYPE has as its value a mask indicating the drawable types that can be created with the corresponding GLXFBConfig (the config is said to "support" these drawable types). The valid bit settings are shown in Table 3.2.

For example, a <code>GLXFBConfig</code> for which the value of the <code>GLX_DRAWABLE_TYPE</code> attribute is

```
GLX_WINDOW_BIT | GLX_PIXMAP_BIT | GLX_PBUFFER_BIT
```

can be used to create any type of GLX drawable, while a GLXFBConfig for which this attribute value is GLX_WINDOW_BIT can not be used to create a GLXPixmap or a GLXPbuffer.

GLX_X_RENDERABLE is a boolean indicating whether X can be used to render into a drawable created with the GLXFBConfig. This attribute is True if the GLXFBConfig supports GLX windows and/or pixmaps.

If a GLXFBConfig supports windows then it has an associated X Visual. The value of the GLX_VISUAL_ID attribute specifies the XID of the Visual and the value of the GLX_X_VISUAL_TYPE attribute specifies the type of Visual. The possible values are shown in Table 3.3. If a GLXFBConfig does not support windows, then querying GLX_VISUAL_ID will return 0 and querying GLX_X_VISUAL_TYPE will return GLX_NONE.

Note that RGBA rendering may be supported for any of the six Visual

types but color index rendering is supported only for PseudoColor, StaticColor, GrayScale, and StaticGray visuals (i.e., single-channel visuals). If RGBA rendering is supported for a single-channel visual (i.e., if the GLX_RENDER_TYPE attribute has the GLX_RGBA_BIT set), then the red component maps to the color buffer bits corresponding to the core X11 visual. The green and blue components map to non-displayed color buffer bits and the alpha component maps to non-displayed alpha buffer bits if their sizes are nonzero, otherwise they are discarded.

The GLX_CONFIG_CAVEAT attribute may be set to one of the following values: GLX_NONE, GLX_SLOW_CONFIG or GLX_NON_CONFORMANT_CONFIG. If the attribute is set to GLX_NONE then the configuration has no caveats; if it is set to GLX_SLOW_CONFIG then rendering to a drawable with this configuration may run at reduced performance (for example, the hardware may not support the color buffer depths described by the configuration); if it is set to GLX_NON_CONFORMANT_CONFIG then rendering to a drawable with this configuration ratio will not pass the required OpenGL conformance tests.

Servers are required to export at least one GLXFBConfig that supports RGBA rendering to windows and passes OpenGL conformance (i.e., the GLX_RENDER_TYPE attribute must have the GLX_RGBA_BIT set, the GLX_DRAWABLE_TYPE attribute must have the GLX_WINDOW_BIT set and the GLX_CONFIG_CAVEAT attribute must not be set to GLX_NON_CONFORMANT_CONFIG). This GLXFBConfig must have at least one color buffer, a stencil buffer of at least 1 bit, a depth buffer of at least 12 bits, and an accumulation buffer; auxillary buffers are optional, and the alpha buffer may have 0 bits. The color buffer size for this GLXFBConfig must be as large as that of the deepest TrueColor, DirectColor, PseudoColor, or StaticColor visual supported on framebuffer level zero (the main image planes), and this configuration must be available on framebuffer level zero.

If the X server exports a PseudoColor or StaticColor visual on framebuffer level 0, a GLXFBConfig that supports color index rendering to windows and passes OpenGL conformance is also required (i.e., the GLX_RENDER_TYPE attribute must have the GLX_COLOR_INDEX_BIT set, the GLX_DRAWABLE_TYPE attribute must have the GLX_WINDOW_BIT set, and the GLX_CONFIG_CAVEAT attribute must not be set to GLX_NON_CONFORMANT_CONFIG). This GLXFBConfig must have at least one color buffer, a stencil buffer of at least 1 bit, and a depth buffer of at least 12 bits. It also must have as many color bitplanes as the deepest PseudoColor or StaticColor visual supported on framebuffer level zero, and the configuration must be made available on level zero.

The attribute GLX_TRANSPARENT_TYPE indicates whether or not the configuration supports transparency, and if it does support transparency, what type of transparency is available. If the attribute is set to GLX_NONE then windows created with the GLXFBConfig will not have any transparent pixels. If the attribute is GLX_TRANSPARENT_RGB or GLX_TRANSPARENT_INDEX then the GLXFBConfig supports transparency. GLX_TRANSPARENT_RGB is only applicable if the configuration is associated with a TrueColor or DirectColor visual: a transparent pixel will be drawn when the red, green and blue values which are read from the framebuffer are equal to GLX_TRANSPARENT_RED_VALUE, GLX_TRANSPARENT_GREEN_VALUE and GLX_TRANSPARENT_BLUE_VALUE, respectively. If the configuration is associated with a PseudoColor, StaticColor, GrayScale or StaticGray visual the transparency mode GLX_TRANSPARENT_INDEX is used. In this case, a transparent pixel will be drawn when the value that is read from the framebuffer is equal to GLX_TRANSPARENT_INDEX_VALUE.

GLX_TRANSPARENT_TYPE is If GLX_NONE or GLX_TRANSPARENT_RGB. then the value for GLX_TRANSPARENT_INDEX_VALUE is undefined. Tf GLX_TRANSPARENT_TYPE is GLX_NONE or GLX_TRANSPARENT_INDEX, then the values for GLX_TRANSPARENT_RED_VALUE, GLX_TRANSPARENT_GREEN_VALUE. and GLX_TRANSPARENT_BLUE_VALUE are undefined. When defined. GLX_TRANSPARENT_GREEN_VALUE, GLX_TRANSPARENT_RED_VALUE, and GLX_TRANSPARENT_BLUE_VALUE are integer framebuffer values between 0 and the maximum framebuffer value for the component. For example, GLX_TRANSPARENT_RED_VALUE will range between 0 and (2**GLX_RED_SIZE)-1. (GLX_TRANSPARENT_ALPHA_VALUE is for future use.)

GLX_MAX_PBUFFER_WIDTH and GLX_MAX_PBUFFER_HEIGHT indicate the maximum width and height that can be passed into glXCreatePbuffer and GLX_MAX_PBUFFER_PIXELS indicates the maximum number of pixels (width times height) for a GLXPbuffer. Note that an implementation may return a value for GLX_MAX_PBUFFER_PIXELS that is less than the maximum width times the maximum height. Also, the value for GLX_MAX_PBUFFER_PIXELS is static and assumes that no other pbuffers or X resources are contending for the framebuffer memory. Thus it may not be possible to allocate a pbuffer of the size given by GLX_MAX_PBUFFER_PIXELS.

Use

GLXFBConfig *glXGetFBConfigs(Display *dpy, int screen, int *nelements);

to get the list of all GLXFBConfigs that are available on the specified screen. The call returns an array of GLXFBConfigs; the number of elements in the array is returned in *nelements*.

Use

GLXFBConfig *glXChooseFBConfig(Display *dpy, int screen, const int *attrib_list, int *nelements);

to get GLXFBConfigs that match a list of attributes.

This call returns an array of GLXFBConfigs that match the specified attributes (attributes are described in Table 3.1). The number of elements in the array is returned in *nelements*.

If $attrib_list$ contains an undefined GLX attribute, *screen* is invalid, or dpy does not support the GLX extension, then NULL is returned.

All attributes in *attrib_list*, including boolean attributes, are immediately followed by the corresponding desired value. The list is terminated with None. If an attribute is not specified in *attrib_list*, then the default value (listed in Table 3.4) is used (it is said to be specified implicitly). For example, if GLX_STEREO is not specified then it is assumed to be False. If GLX_DONT_CARE is specified as an attribute value, then the attribute will not be checked. GLX_DONT_CARE may be specified for all attributes except GLX_LEVEL. If *attrib_list* is NULL or empty (first attribute is None), then selection and sorting of GLXFBConf igs is done according to the default criteria in Tables 3.4 and 3.1, as described below under Selection and Sorting.

Selection of GLXFBConfigs

Attributes are matched in an attribute-specific manner, as shown in Table 3.4. The match criteria listed in the table have the following meanings:

- Smaller GLXFBConfigs with an attribute value that meets or exceeds the specified value are returned.
- Larger GLXFBConfigs with an attribute value that meets or exceeds the specified value are returned.
- *Exact* Only GLXFBConfigs whose attribute value exactly matches the requested value are considered.
- Mask Only GLXFBConfigs for which the set bits of attribute include all the bits that are set in the requested value are considered. (Additional bits might be set in the attribute).

Some of the attributes, such as GLX_LEVEL, must match the specified value exactly; others, such as GLX_RED_SIZE must meet or exceed the specified minimum values.

To retrieve an GLXFBConfig given its XID, use the GLX_FBCONFIG_ID attribute. When GLX_FBCONFIG_ID is specified, all other attributes are ignored, and only the GLXFBConfig with the given XID is returned (NULL is returned if it does not exist).

If GLX_MAX_PBUFFER_WIDTH, GLX_MAX_PBUFFER_HEIGHT, GLX_MAX_PBUFFER_PIXELS, or GLX_VISUAL_ID are specified in *attrib_list*, then they are ignored (however, if present, these attributes must still be followed by an attribute value in *attrib_list*). If GLX_DRAWABLE_TYPE is specified in *attrib_list* and the mask that follows does not have GLX_WINDOW_BIT set, then the GLX_X_VISUAL_TYPE attribute is ignored.

If GLX_TRANSPARENT_TYPE is set to GLX_NONE in *attrib_list*, then inclusion of GLX_TRANSPARENT_INDEX_VALUE, GLX_TRANSPARENT_RED_VALUE, GLX_TRANSPARENT_GREEN_VALUE, GLX_TRANSPARENT_BLUE_VALUE, or GLX_TRANSPARENT_ALPHA_VALUE will be ignored.

If no GLXFBConfig matching the attribute list exists, then NULL is returned. If exactly one match is found, a pointer to that GLXFBConfig is returned.

Sorting of GLXFBConfigs

If more than one matching GLXFBConfig is found, then a list of GLXFBConfigs, sorted according to the *best* match criteria, is returned. The list is sorted according to the following precedence rules that are applied in ascending order (i.e., configurations that are considered equal by lower numbered rule are sorted by the higher numbered rule):

- 1. By GLX_CONFIG_CAVEAT where the precedence is GLX_NONE, GLX_SLOW_CONFIG, GLX_NON_CONFORMANT_CONFIG.
- 2. Larger total number of RGBA color bits (GLX_RED_SIZE, GLX_GREEN_SIZE, GLX_BLUE_SIZE, plus GLX_ALPHA_SIZE). If the requested number of bits in *attrib_list* for a particular color component is 0 or GLX_DONT_CARE, then the number of bits for that component is not considered.
- 3. Smaller GLX_BUFFER_SIZE.
- 4. Single buffered configuration (GLX_DOUBLE_BUFFER being False) precedes a double buffered one.
- 5. Smaller GLX_AUX_BUFFERS.

Attribute	Default	Selection	Sort
		and Sorting	Priority
		$\operatorname{Criteria}$	
GLX_FBCONFIG_ID	GLX_DONT_CARE	Exact	
GLX_BUFFER_SIZE	0	Smaller	3
GLX_LEVEL	0	Exact	
GLX_DOUBLEBUFFER	GLX_DONT_CARE	Exact	4
GLX_STERE0	False	Exact	
GLX_AUX_BUFFERS	0	Smaller	5
GLX_RED_SIZE	0	Larger	2
GLX_GREEN_SIZE	0	Larger	2
GLX_BLUE_SIZE	0	Larger	2
GLX_ALPHA_SIZE	0	Larger	2
GLX_DEPTH_SIZE	0	Larger	6
GLX_STENCIL_SIZE	0	Larger	7
GLX_ACCUM_RED_SIZE	0	Larger	8
GLX_ACCUM_GREEN_SIZE	0	Larger	8
GLX_ACCUM_BLUE_SIZE	0	Larger	8
GLX_ACCUM_ALPHA_SIZE	0	Larger	8
GLX_RENDER_TYPE	GLX_RGBA_BIT	Mask	
GLX_DRAWABLE_TYPE	GLX_WINDOW_BIT	Mask	
GLX_X_RENDERABLE	GLX_DONT_CARE	Exact	
GLX_X_VISUAL_TYPE	GLX_DONT_CARE	Exact	9
GLX_CONFIG_CAVEAT	GLX_DONT_CARE	Exact	1
GLX_TRANSPARENT_TYPE	GLX_NONE	Exact	
GLX_TRANSPARENT_INDEX_VALUE	GLX_DONT_CARE	Exact	
GLX_TRANSPARENT_RED_VALUE	GLX_DONT_CARE	Exact	
GLX_TRANSPARENT_GREEN_VALUE	GLX_DONT_CARE	Exact	
GLX_TRANSPARENT_BLUE_VALUE	GLX_DONT_CARE	Exact	
GLX_TRANSPARENT_ALPHA_VALUE	GLX_DONT_CARE	Exact	

Table 3.4: Default values and match criteria for GLXFBConfig attributes.

- 6. Larger GLX_DEPTH_SIZE.
- 7. Smaller GLX_STENCIL_BITS.
- 8. Larger total number of accumulation buffer color bits (GLX_ACCUM_RED_SIZE, GLX_ACCUM_GREEN_SIZE, GLX_ACCUM_BLUE_SIZE, plus GLX_ACCUM_ALPHA_SIZE). If the requested number of bits in *attrib_list* for a particular color component is 0 or GLX_DONT_CARE, then the number of bits for that component is not considered.
- 9. By GLX_X_VISUAL_TYPE where the precedence is GLX_TRUE_COLOR, GLX_DIRECT_COLOR, GLX_PSEUDO_COLOR, GLX_STATIC_COLOR, GLX_GRAY_SCALE, GLX_STATIC_GRAY.

Use **XFree** to free the memory returned by **glXChooseFBConfig**. To get the value of a GLX attribute for a **GLXFBConfig** use

If glXGetFBConfigAttrib succeeds then it returns Success and the value for the specified attribute is returned in *value*; otherwise it returns one of the following errors:

GLX_BAD_ATTRIBUTE attribute is not a valid GLX attribute.

Refer to Table 3.1 and Table 3.4 for a list of valid GLX attributes.

A GLXFBConfig has an associated X Visual only if the GLX_DRAWABLE_TYPE attribute has the GLX_WINDOW_BIT bit set. To retrieve the associated visual, call:

XVisualInfo *glXGetVisualFromFBConfig(Display *dpy, GLXFBConfig config);

If config is a valid GLXFBConfig and it has an associated X visual then information describing that visual is returned; otherwise NULL is returned. Use **XFree** to free the data returned.

3.3.4 On Screen Rendering

To create an onscreen rendering area, first create an X Window with a visual that corresponds to the desired GLXFBConfig, then call

```
GLXWindow glXCreateWindow(Display *dpy,
GLXFBConfig config, Window win, const int
*attrib_list);
```

glXCreateWindow creates a GLXWindow and returns its XID. Any GLX rendering context created with a compatible GLXFBConfig can be used to render into this window.

attrib_list specifies a list of attributes for the window. The list has the same structure as described for **glXChooseFBConfig**. Currently no attributes are recognized, so *attrib_list* must be NULL or empty (first attribute of None).

If win was not created with a visual that corresponds to config, then a BadMatch error is generated. (i.e., glXGetVisualFromFBConfig must return the visual corresponding to win when the GLXFBConfig parameter is set to config.) If config does not support rendering to windows (the GLX_DRAWABLE_TYPE attribute does not contain GLX_WINDOW_BIT), a BadMatch error is generated. If config is not a valid GLXFBConfig, a GLXBadFBConfig error is generated. If win is not a valid window XID, then a BadWindow error is generated. If there is already a GLXFBConfig associated with win (as a result of a previous glXCreateWindow call), then a BadAlloc error is generated. Finally, if the server cannot allocate the new GLX window, a BadAlloc error is generated.

A GLXWindow is destroyed by calling

```
glXDestroyWindow(Display *dpy, GLXWindow win);
```

This request deletes the association between the resource ID *win* and the GLX window. The storage will be freed when it is not current to any client.

If win is not a valid GLX window then a GLXBadWindow error is generated.

3.3.5 Off Screen Rendering

GLX supports two types of offscreen rendering surfaces: GLXPixmaps and GLXPbuffers. GLXPixmaps and GLXPbuffers differ in the following ways:

1. GLXPixmaps have an associated X pixmap and can therefore be rendered to by X. Since a GLXPbuffer is a GLX resource, it may not be possible to render to it using X or an X extension other than GLX.

- 2. The format of the color buffers and the type and size of any associated ancillary buffers for a GLXPbuffer can only be described with a GLXFBConfig. The older method of using extended X Visuals to describe the configuration of a GLXDrawable cannot be used. (See section 3.4 for more information on extended visuals.)
- 3. It is possible to create a GLXPbuffer whose contents may be asynchronously lost at any time.
- 4. If the GLX implementation supports direct rendering, then it must support rendering to GLXPbuffers via a direct rendering context. Although some implementations may support rendering to GLXPixmaps via a direct rendering context, GLX does not require this to be supported.
- 5. The intent of the pbuffer semantics is to enable implementations to allocate pbuffers in non-visible frame buffer memory. Thus, the allocation of a GLXPbuffer can fail if there is insufficient framebuffer resources. (Implementations are not required to virtualize pbuffer memory.) Also, clients should deallocate GLXPbuffers when they are no longer using them for example, when the program is iconified.

To create a GLXPixmap offscreen rendering area, first create an X Pixmap of the depth specified by the desired GLXFBConfig, then call

glXCreatePixmap creates an offscreen rendering area and returns its XID. Any GLX rendering context created with a GLXFBConfig that is compatible with *config* can be used to render into this offscreen area.

pixmap is used for the RGB planes of the front-left buffer of the resulting GLX offscreen rendering area. GLX pixmaps may be created with a *config* that includes back buffers and stereoscopic buffers. However, **glXSwap-Buffers** is ignored for these pixmaps.

attrib_list specifies a list of attributes for the pixmap. The list has the same structure as described for glXChooseFBConfig. Currently no attributes are recognized, so attrib_list must be NULL or empty (first attribute of None).

A direct rendering context might not be able to be made current with a GLXPixmap.

If pixmap was not created with respect to the same screen as config, then a BadMatch error is generated. If config is not a valid GLXFBConfig or if it does not support pixmap rendering then a GLXBadFBConfig error is generated. If pixmap is not a valid Pixmap XID, then a BadPixmap error is generated. Finally, if the server cannot allocate the new GLX pixmap, a BadAlloc error is generated.

A GLXPixmap is destroyed by calling

glXDestroyPixmap(Display *dpy, GLXPixmap pixmap);

This request deletes the association between the XID *pixmap* and the GLX pixmap. The storage for the GLX pixmap will be freed when it is not current to any client. To free the associated X pixmap, call **XFreePixmap**.

If *pixmap* is not a valid GLX pixmap then a GLXBadPixmap error is generated.

To create a GLXPbuffer call

```
GLXPbuffer glXCreatePbuffer(Display *dpy,
GLXFBConfig config, const int *attrib_list);
```

This creates a single GLXPbuffer and returns its XID. Like other drawable types, GLXPbuffers are shared; any client which knows the associated XID can use a GLXPbuffer.

attrib_list specifies a list of attributes for the pbuffer. The list has the same structure as described for glXChooseFBConfig. Currently only four attributes can be specified in *attrib_list*: GLX_PBUFFER_WIDTH, GLX_PBUFFER_HEIGHT, GLX_PRESERVED_CONTENTS and GLX_LARGEST_PBUFFER.

attrib_list may be NULL or empty (first attribute of None), in which case all the attributes assume their default values as described below.

GLX_PBUFFER_WIDTH and GLX_PBUFFER_HEIGHT specify the pixel width and height of the rectangular pbuffer. The default values for GLX_PBUFFER_WIDTH and GLX_PBUFFER_HEIGHT are zero.

Use GLX_LARGEST_PBUFFER to get the largest available pbuffer when the allocation of the pbuffer would otherwise fail. The width and height of the allocated pbuffer will never exceed the values of GLX_PBUFFER_WIDTH and GLX_PBUFFER_HEIGHT, respectively. Use glXQueryDrawable to retrieve the dimensions of the allocated pbuffer. By default, GLX_LARGEST_PBUFFER is False.

If the GLX_PRESERVED_CONTENTS attribute is set to False in *attrib_list*, then an *unpreserved* pbuffer is created and the contents of the pbuffer may be lost at any time. If this attribute is not specified, or if it is specified as **True** in *attrib_list*, then when a resource conflict occurs the contents of the pbuffer will be *preserved* (most likely by swapping out portions of the buffer from the framebuffer to main memory). In either case, the client can register to receive a pbuffer clobber event which is generated when the pbuffer contents have been preserved or have been damaged. (See **glXSelectEvent** in section 3.3.8 for more information.)

The resulting pbuffer will contain color buffers and ancillary buffers as specified by *config*. It is possible to create a pbuffer with back buffers and to swap the front and back buffers by calling **glXSwapBuffers**. Note that pbuffers use framebuffer resources so applications should consider deallocating them when they are not in use.

If a pbuffer is created with GLX_PRESERVED_CONTENTS set to False, then portions of the buffer contents may be lost at any time due to frame buffer resource conflicts. Once the contents of a unpreserved pbuffer have been lost it is considered to be in a *damaged* state. It is not an error to render to a pbuffer that is in this state but the effect of rendering to it is the same as if the pbuffer were destroyed: the context state will be updated, but the frame buffer state becomes undefined. It is also not an error to query the pixel contents of such a pbuffer, but the values of the returned pixels are undefined. Note that while this specification allows for unpreserved pbuffers to be damaged as a result of other pbuffer activity, the intent is to have only the activity of visible windows damage pbuffers.

Since the contents of a unpreserved pbuffer can be lost at anytime with only asynchronous notification (via the pbuffer clobber event), the only way a client can guarantee that valid pixels are read back with **glReadPixels** is by grabbing the X server. (Note that this operation is potentially expensive and should not be done frequently. Also, since this locks out other X clients, it should be done only for short periods of time.) Clients that don't wish to do this can check if the data returned by **glReadPixels** is valid by calling **XSync** and then checking the event queue for pbuffer clobber events (assuming that these events had been pulled off of the queue prior to the **glReadPixels** call).

When **glXCreatePbuffer** fails to create a **GLXPbuffer** due to insufficient resources, a **BadAlloc** error is generated. If *config* is not a valid **GLXFBConfig** then a **GLXBadFBConfig** error is generated; if *config* does not support **GLXPbuffers** then a **BadMatch** error is generated.

A GLXPbuffer is destroyed by calling:

```
void glXDestroyPbuffer(Display *dpy, GLXPbuffer
    pbuf);
```

The XID associated with the GLXPbuffer is destroyed. The storage for the GLXPbuffer will be destroyed once it is no longer current to any client.

If pbuf is not a valid GLXPbuffer then a GLXBadPbuffer error is generated.

3.3.6 Querying Attributes

To query an attribute associated with a GLXDrawable call:

attribute must be set to one of GLX_WIDTH, GLX_HEIGHT, GLX_PRESERVED_CONTENTS, GLX_LARGEST_PBUFFER, or GLX_FBCONFIG_ID.

To get the GLXFBConfig for a GLXDrawable, first retrieve the XID for the GLXFBConfig and then call glXChooseFBConfig.

If *draw* is not a valid GLXDrawable then a GLXBadDrawable error is generated. If *draw* is a GLXWindow or GLXPixmap and *attribute* is set to GLX_PRESERVED_CONTENTS or GLX_LARGEST_PBUFFER, then the contents of *value* are undefined.

3.3.7 Rendering Contexts

To create an OpenGL rendering context, call

```
GLXContext glXCreateNewContext(Display *dpy,
GLXFBConfig config, int render_type, GLXContext
share_list, Bool direct);
```

glXCreateNewContext returns NULL if it fails. If glXCreateNewContext succeeds, it initializes the rendering context to the initial OpenGL state and returns a handle to it. This handle can be used to render to GLX windows, GLX pixmaps and GLX pbuffers.

If render_type is set to GLX_RGBA_TYPE then a context that supports RGBA rendering is created; if render_type is set to GLX_COLOR_INDEX_TYPE then a context that supports color index rendering is created.

If *share_list* is not NULL, then all display lists and texture objects except texture objects named 0 will be shared by *share_list* and the newly created

rendering context. An arbitrary number of **GLXContexts** can share a single display list and texture object space. The server context state for all sharing contexts must exist in a single address space or a **BadMatch** error is generated.

If *direct* is true, then a direct rendering context will be created if the implementation supports direct rendering and the connection is to an X server that is local. If *direct* is False, then a rendering context that renders through the X server is created.

Direct rendering contexts may be a scarce resource in some implementations. If *direct* is true, and if a direct rendering context cannot be created, then **glXCreateNewContext** will attempt to create an indirect context instead.

glXCreateNewContext can generate the following errors: GLXBadContext if *share_list* is neither zero nor a valid GLX rendering context; GLXBadFBConfig if *config* is not a valid GLXFBConfig; BadMatch if the server context state for *share_list* exists in an address space that cannot be shared with the newly created context or if *share_list* was created on a different screen than the one referenced by *config*; BadAlloc if the server does not have enough resources to allocate the new context; BadValue if *render_type* does not refer to a valid rendering type.

To determine if an OpenGL rendering context is direct, call

```
Bool glXIsDirect(Display *dpy, GLXContext ctx);
```

glXIsDirect returns True if ctx is a direct rendering context, False otherwise. If ctx is not a valid GLX rendering context, a GLXBadContext error is generated.

An OpenGL rendering context is destroyed by calling

If ctx is still current to any thread, ctx is not destroyed until it is no longer current. In any event, the associated XID will be destroyed and ctx cannot subsequently be made current to any thread.

glXDestroyContext will generate a GLXBadContext error if *ctx* is not a valid rendering context.

To make a context current, call

```
Bool glXMakeContextCurrent(Display * dpy,
GLXDrawable draw, GLXDrawable read, GLXContext
ctx);
```

glXMakeContextCurrent binds *ctx* to the current rendering thread and to the *draw* and *read* drawables. *draw* is used for all OpenGL operations except:

- Any pixel data that are read based on the value of GL_READ_BUFFER. Note that accumulation operations use the value of GL_READ_BUFFER, but are not allowed unless *draw* is identical to *read*.
- Any depth values that are retrieved by glReadPixels or glCopyPixels.
- Any stencil values that are retrieved by glReadPixels or glCopyPixels.

These frame buffer values are taken from *read*. Note that the same **GLXDrawable** may be specified for both *draw* and *read*.

If the calling thread already has a current rendering context, then that context is flushed and marked as no longer current. ctx is made the current context for the calling thread.

If draw or read are not compatible with ctx a BadMatch error is generated. If ctx is current to some other thread, then glXMakeContextCurrent will generate a BadAccess error. GLXBadContextState is generated if there is a current rendering context and its render mode is either GL_FEEDBACK or GL_SELECT. If ctx is not a valid GLX rendering context, GLXBadContext is generated. If either draw or read are not a valid GLX drawable, a GLXBadDrawable error is generated. If the X Window underlying either draw or read is no longer valid, a GLXBadWindow error is generated. If the previous context of the calling thread has unflushed commands, and the previous drawable is no longer valid, GLXBadCurrentDrawable is generated. Note that the ancillary buffers for draw and read need not be allocated until they are needed. A BadAlloc error will be generated if the server does not have enough resources to allocate the buffers.

In addition, implementations may generate a BadMatch error under the following conditions: if *draw* and *read* cannot fit into framebuffer memory simultaneously; if *draw* or *read* is a GLXPixmap and *ctx* is a direct rendering context; if *draw* or *read* is a GLXPixmap and *ctx* was previously bound to a GLXWindow or GLXPbuffer; if *draw* or *read* is a GLXPixmap.

Other errors may arise when the context state is inconsistent with the drawable state, as described in the following paragraphs. Color buffers are

treated specially because the current GL_DRAW_BUFFER and GL_READ_BUFFER context state can be inconsistent with the current draw or read drawable (for example, when GL_DRAW_BUFFER is GL_BACK and the drawable is single buffered).

No error will be generated if the value of GL_DRAW_BUFFER in *ctx* indicates a color buffer that is not supported by *draw*. In this case, all rendering will behave as if GL_DRAW_BUFFER was set to NONE. Also, no error will be generated if the value of GL_READ_BUFFER in *ctx* does not correspond to a valid color buffer. Instead, when an operation that reads from the color buffer is executed (e.g., glReadPixels or glCopyPixels), the pixel values used will be undefined until GL_READ_BUFFER is set to a color buffer that is valid in *read*. Operations that query the value of GL_READ_BUFFER or GL_DRAW_BUFFER (i.e., glGet, glPushAttrib) use the value set last in the context, independent of whether it is a valid buffer in *read* or *draw*.

Note that it is an error to later call glDrawBuffer and/or glRead-Buffer (even if they are implicitly called via glPopAttrib or glXCopy-Context) and specify a color buffer that is not supported by *draw* or *read*. Also, subsequent calls to glReadPixels or glCopyPixels that specify an unsupported ancillary buffer will result in an error.

If draw is destroyed after glXMakeContextCurrent is called, then subsequent rendering commands will be processed and the context state will be updated, but the frame buffer state becomes undefined. If *read* is destroyed after glXMakeContextCurrent then pixel values read from the framebuffer (e.g., as result of calling glReadPixels, glCopyPixels or glCopyColorTable) are undefined. If the X Window underlying the GLXWindow draw or read drawable is destroyed, rendering and readback are handled as above.

To release the current context without assigning a new one, set *ctx* to NULL and set *draw* and *read* to None. If *ctx* is NULL and *draw* and *read* are not None, or if *draw* or *read* are set to None and *ctx* is not NULL, then a BadMatch error will be generated.

The first time ctx is made current, the viewport and scissor dimensions are set to the size of the draw drawable (as though glViewport(0, 0, w, h) and glScissor(0, 0, w, h) were called, where w and h are the width and height of the drawable, respectively). However, the viewport and scissor dimensions are not modified when ctx is subsequently made current; it is the clients responsibility to reset the viewport and scissor in this case.

Note that when multiple threads are using their current contexts to render to the same drawable, OpenGL does not guarantee atomicity of fragment update operations. In particular, programmers may not assume that depth-buffering will automatically work correctly; there is a race condition between threads that read and update the depth buffer. Clients are responsible for avoiding this condition. They may use vendor-specific extensions or they may arrange for separate threads to draw in disjoint regions of the framebuffer, for example.

To copy OpenGL rendering state from one context to another, use

void glXCopyContext(Display *dpy, GLXContext source, GLXContext dest, unsigned long mask);

glXCopyContext copies selected groups of state variables from *source* to *dest. mask* indicates which groups of state variables are to be copied; it contains the bitwise OR of the symbolic names for the attribute groups. The symbolic names are the same as those used by glPushAttrib, described in the OpenGL Specification. Also, the order in which the attributes are copied to *dest* as a result of the glXCopyContext operation is the same as the order in which they are popped off of the stack when glPopAttrib is called. The single symbolic constant GL_ALL_ATTRIB_BITS can be used to copy the maximum possible portion of the rendering state. It is not an error to specify *mask* bits that are undefined.

Not all GL state values can be copied. For example, client side state such as pixel pack and unpack state, vertex array state and select and feedback state cannot be copied. Also, some server state such as render mode state, the contents of the attribute and matrix stacks, display lists and texture objects, cannot be copied. The state that can be copied is exactly the state that is manipulated by **glPushAttrib**.

If source and dest were not created on the same screen or if the server context state for source and dest does not exist in the same address space, a BadMatch error is generated (source and dest may be based on different GLXFBConfigs and still share an address space; glXCopyContext will work correctly in such cases). If the destination context is current for some thread then a BadAccess error is generated. If the source context is the same as the current context of the calling thread, and the current drawable of the calling thread is no longer valid, a GLXBadCurrentDrawable error is generated. Finally, if either source or dest is not a valid GLX rendering context, a GLXBadContext error is generated.

glXCopyContext performs an implicit **glFlush** if *source* is the current context for the calling thread.

Only one rendering context may be in use, or *current*, for a particular thread at a given time. The minimum number of current rendering contexts that must be supported by a GLX implementation is one. (Supporting a

Attribute	Type	Description	
GLX_FBCONFIG_ID	XID	XID of GLXFBConfig associated with context	
GLX_RENDER_TYPE	int	type of rendering supported	
GLX_SCREEN	int	screen number	



larger number of current rendering contexts is essential for general-purpose systems, but may not be necessary for turnkey applications.)

To get the current context, call

```
GLXContext glXGetCurrentContext(void);
```

If there is no current context, NULL is returned.

To get the XID of the current drawable used for rendering, call

```
GLXDrawable glXGetCurrentDrawable(void);
```

If there is no current *draw* drawable, None is returned. To get the XID of the current drawable used for reading, call

```
GLXDrawable glXGetCurrentReadDrawable(void);
```

If there is no current *read* drawable, **None** is returned. To get the display associated with the current context and drawable, call

```
Display *glXGetCurrentDisplay(void);
```

If there is no current context, NULL is returned.

To obtain the value of a context's attribute, use

glXQueryContext returns through *value* the value of *attribute* for *ctx*. It may cause a round trip to the server.

The values and types corresponding to each GLX context attribute are listed in Table 3.5.

glXQueryContext returns GLX_BAD_ATTRIBUTE if *attribute* is not a valid GLX context attribute and Success otherwise. If *ctx* is invalid and a round trip to the server is involved, a GLXBadContext error is generated.

glXGet* calls retrieve client-side state and do not force a round trip to the X server. Unlike most X calls (including the **glXQuery*** calls) that return a value, these calls do not flush any pending requests.

3.3.8 Events

GLX events are returned in the X11 event stream. GLX and X11 events are selected independently; if a client selects for both, then both may be delivered to the client. The relative order of X11 and GLX events is not specified.

A client can ask to receive GLX events on a GLXWindow or a GLXPbuffer by calling

void glXSelectEvent(Display *dpy, GLXDrawable draw, unsigned long event_mask);

Calling **glXSelectEvent** overrides any previous event mask that was set by the client for *draw*. Note that the GLX event mask is private to GLX (separate from the core X11 event mask), and that a separate GLX event mask is maintained in the server state for each client for each drawable.

If *draw* is not a valid GLXPbuffer or a valid GLXWindow, a GLXBadDrawable error is generated.

To find out which GLX events are selected for a $\operatorname{GLXWindow}$ or $\operatorname{GLXPbuffer} \operatorname{call}$

If *draw* is not a GLX window or pbuffer then a GLXBadDrawable error is generated.

Currently only one GLX event can be selected, by setting *event_mask* to GLX_PBUFFER_CLOBBER_MASK. The data structure describing a pbuffer clobber event is:

```
typedef struct {
    int event_type; /* GLX_DAMAGED or GLX_SAVED */
    int draw_type; /* GLX_WINDOW or GLX_PBUFFER */
    unsigned long serial; /* number of last request processed by server */
    Bool send_event; /* event was generated by a SendEvent request */
    Display *display; /* display the event was read from */
    GLXDrawable drawable; /* XID of Drawable */
    unsigned int buffer_mask; /* mask indicating which buffers are affected */
    unsigned int aux_buffer; /* which aux buffer was affected */
    int x, y;
    int width, height;
```

Bitmask	Corresponding buffer
GLX_FRONT_LEFT_BUFFER_BIT	Front left color buffer
GLX_FRONT_RIGHT_BUFFER_BIT	Front right color buffer
GLX_BACK_LEFT_BUFFER_BIT	Back left color buffer
GLX_BACK_RIGHT_BUFFER_BIT	Back right color buffer
GLX_AUX_BUFFERS_BIT	Auxillary buffer
GLX_DEPTH_BUFFER_BIT	Depth buffer
GLX_STENCIL_BUFFER_BIT	Stencil buffer
GLX_ACCUM_BUFFER_BIT	Accumulation buffer

Table 3.6: Masks identifying clobbered buffers.

int count; /* if nonzero, at least this many more */
} GLXPbufferClobberEvent;

If an implementation doesn't support the allocation of pbuffers, then it doesn't need to support the generation of GLXPbufferClobberEvents.

A single X server operation can cause several pbuffer clobber events to be sent (e.g., a single pbuffer may be damaged and cause multiple pbuffer clobber events to be generated). Each event specifies one region of the GLXDrawable that was affected by the X Server operation. *buffer_mask* indicates which color or ancillary buffers were affected; the bits that may be present in the mask are listed in Table 3.6. All the pbuffer clobber events generated by a single X server action are guaranteed to be contiguous in the event queue. The conditions under which this event is generated and the value of *event_type* varies, depending on the type of the GLXDrawable.

When the GLX_AUX_BUFFERS_BIT is set in *buffer_mask*, then *aux_buffer* is set to indicate which buffer was affected. If more than one aux buffer was affected, then additional events are generated as part of the same contiguous event group. Each additional event will have only the GLX_AUX_BUFFERS_BIT set in *buffer_mask*, and the *aux_buffer* field will be set appropriately. For non-stereo drawables, GLX_FRONT_LEFT_BUFFER_BIT and GLX_BACK_LEFT_BUFFER_BIT are used to specify the front and back color buffers.

For preserved pbuffers, a pbuffer clobber event, with *event_type* GLX_SAVED, is generated whenever the contents of a pbuffer has to be moved to avoid being damaged. The event(s) describes which portions of the pbuffer were affected. Clients who receive many pbuffer clobber events, referring to different save actions, should consider freeing the pbuffer resource in order

to prevent the system from thrashing due to insufficient resources.

For an unpreserved pbuffer a pbuffer clobber event, with *event_type* GLX_DAMAGED, is generated whenever a portion of the pbuffer becomes invalid.

For GLX windows, pbuffer clobber events with *event_type* GLX_SAVED occur whenever an ancillary buffer, associated with the window, gets moved out of offscreen memory. The event contains information indicating which color or ancillary buffers, and which portions of those buffers, were affected. GLX windows don't generate pbuffer clobber events when clobbering each others' ancillary buffers, only standard X11 damage events

3.3.9 Synchronization Primitives

To prevent X requests from executing until any outstanding OpenGL rendering is done, call

```
void glXWaitGL(void);
```

OpenGL calls made prior to glXWaitGL are guaranteed to be executed before X rendering calls made after glXWaitGL. While the same result can be achieved using glFinish, glXWaitGL does not require a round trip to the server, and is therefore more efficient in cases where the client and server are on separate machines.

glXWaitGL is ignored if there is no current rendering context. If the drawable associated with the calling thread's current context is no longer valid, a GLXBadCurrentDrawable error is generated.

To prevent the OpenGL command sequence from executing until any outstanding X requests are completed, call

```
void glXWaitX(void);
```

X rendering calls made prior to glXWaitX are guaranteed to be executed before OpenGL rendering calls made after glXWaitX. While the same result can be achieved using XSync, glXWaitX does not require a round trip to the server, and may therefore be more efficient.

glXWaitX is ignored if there is no current rendering context. If the drawable associated with the calling thread's current context is no longer valid, a GLXBadCurrentDrawable error is generated.

3.3.10 Double Buffering

For drawables that are double buffered, the contents of the back buffer can be made potentially visible (i.e., become the contents of the front buffer) by calling

void glXSwapBuffers(Display *dpy, GLXDrawable draw);

The contents of the back buffer then become undefined. This operation is a no-op if *draw* was created with a non-double-buffered GLXFBConfig, or if *draw* is a GLXPixmap.

All GLX rendering contexts share the same notion of which are front buffers and which are back buffers for a given drawable. This notion is also shared with the X double buffer extension (DBE).

When multiple threads are rendering to the same drawable, only one of them need call **glXSwapBuffers** and all of them will see the effect of the swap. The client must synchronize the threads that perform the swap and the rendering, using some means outside the scope of GLX, to insure that each new frame is completely rendered before it is made visible.

If dpy and draw are the display and drawable for the calling thread's current context, **glXSwapBuffers** performs an implicit **glFlush**. Subsequent OpenGL commands can be issued immediately, but will not be executed until the buffer swapping has completed, typically during vertical retrace of the display monitor.

If draw is not a valid GLX drawable, glXSwapBuffers generates a GLXBadDrawable error. If dpy and draw are the display and drawable associated with the calling thread's current context, and if draw is a window that is no longer valid, a GLXBadCurrentDrawable error is generated. If the X Window underlying draw is no longer valid, a GLXBadWindow error is generated.

3.3.11 Access to X Fonts

A shortcut for using X fonts is provided by the command

count display lists are defined starting at *list_base*, each list consisting of a single call on **glBitmap**. The definition of bitmap *list_base* + i is taken from the glyph *first* + i of *font*. If a glyph is not defined, then an empty display list is constructed for it. The width, height, xorig, and yorig of the constructed bitmap are computed from the font metrics as rbearing-lbearing, ascent+descent, -lbearing, and descent respectively. xmove is taken from the width metric and ymove is set to zero.

Note that in the direct rendering case, this requires that the bitmaps be copied to the client's address space.

glXUseXFont performs an implicit glFlush.

glXUseXFont is ignored if there is no current GLX rendering context. BadFont is generated if *font* is not a valid X font id. GLXBadContextState is generated if the current GLX rendering context is in display list construction mode. GLXBadCurrentDrawable is generated if the drawable associated with the calling thread's current context is no longer valid.

3.4 Backwards Compatibility

GLXFBConfigs were introduced in GLX 1.3. Also, new functions for managing drawable configurations, creating pixmaps, destroying pixmaps, creating contexts and making a context current were introduced. The 1.2 versions of these functions are still available and are described in this section. Even though these older function calls are supported their use is not recommended.

3.4.1 Using Visuals for Configuration Management

In order to maintain backwards compatibility, visuals continue to be overloaded with information describing the ancillary buffers and color buffers for GLXPixmaps and Windows. Note that Visuals cannot be used to create GLXPbuffers. Also, not all configuration attributes are exported through visuals (e.g., there is no visual attribute to describe which drawables are supported by the visual.)

The set of extended Visuals is fixed at server start up time. Thus a server can export multiple Visuals that differ only in the extended attributes. Implementors may choose to export fewer GLXDrawable configurations through visuals than through GLXFBConfigs.

The X protocol allows a single VisualID to be instantiated at multiple depths. Since GLX allows only one depth for any given VisualID, an XVisualInfo is used by GLX functions. An XVisualInfo is a {Visual, Screen, Depth} triple and can therefore be interpreted unambiguously.

The constants shown in Table 3.7 are passed to glXGetConfig and glXChooseVisual to specify which attributes are being queried.

To obtain a description of an OpenGL attribute exported by a Visual use

Attribute	Type	Notes
GLX_USE_GL	boolean	True if OpenGL rendering supported
GLX_BUFFER_SIZE	integer	depth of the color buffer
GLX_LEVEL	integer	frame buffer level
GLX_RGBA	boolean	True if RGBA rendering supported
GLX_DOUBLEBUFFER	boolean	True if color buffers have front/back pairs
GLX_STEREO	boolean	True if color buffers have left/right pairs
GLX_AUX_BUFFERS	integer	number of auxiliary color buffers
GLX_RED_SIZE	integer	number of bits of Red in the color buffer
GLX_GREEN_SIZE	integer	number of bits of Green in the color buffer
GLX_BLUE_SIZE	integer	number of bits of Blue in the color buffer
GLX_ALPHA_SIZE	integer	number of bits of Alpha in the color buffer
GLX_DEPTH_SIZE	integer	number of bits in the depth buffer
GLX_STENCIL_SIZE	integer	number of bits in the stencil buffer
GLX_ACCUM_RED_SIZE	integer	number Red bits in the accumulation buffer
GLX_ACCUM_GREEN_SIZE	integer	number Green bits in the accumulation buffer
GLX_ACCUM_BLUE_SIZE	integer	number Blue bits in the accumulation buffer
GLX_ACCUM_ALPHA_SIZE	integer	number Alpha bits in the accumulation buffer
GLX_FBCONFIG_ID	integer	XID of most closely associated GLXFBConfig

Table 3.7: GLX attributes for Visuals.

glXGetConfig returns through *value* the value of the *attribute* of *visual*. glXGetConfig returns one of the following error codes if it fails, and Success otherwise:

GLX_NO_EXTENSION dpy does not support the GLX extension.

GLX_BAD_SCREEN screen of visual does not correspond to a screen.

GLX_BAD_ATTRIBUTE attribute is not a valid GLX attribute.

GLX_BAD_VISUAL visual does not support GLX and an attribute other than GLX_USE_GL was specified.

GLX_BAD_VALUE parameter invalid

A GLX implementation may export many visuals that support OpenGL. These visuals support either color index or RGBA rendering. RGBA rendering can be supported only by Visuals of type TrueColor or DirectColor (unless GLXFBConfigs are used), and color index rendering can be supported only by Visuals of type PseudoColor or StaticColor.

glXChooseVisual is used to find a visual that matches the client's specified attributes.

XVisualInfo *glXChooseVisual(Display *dpy, int screen, int *attrib_list);

glXChooseVisual returns a pointer to an XVisualInfo structure describing the visual that best matches the specified attributes. If no matching visual exists, NULL is returned.

The attributes are matched in an attribute-specific manner, as shown in Table 3.8. The definitions for the selection criteria Smaller, Larger, and Exact are given in section 3.3.3.

If GLX_RGBA is in *attrib_list* then the resulting visual will be TrueColor or DirectColor. If all other attributes are equivalent, then a TrueColor visual will be chosen in preference to a DirectColor visual.

If GLX_RGBA is not in *attrib_list* then the returned visual will be PseudoColor or StaticColor. If all other attributes are equivalent then a PseudoColor visual will be chosen in preference to a StaticColor visual.

Attribute	Default	Selection Criteria
GLX_USE_GL	True	Exact
GLX_BUFFER_SIZE	0	Smaller
GLX_LEVEL	0	Exact
GLX_RGBA	False	Exact
GLX_DOUBLEBUFFER	False	Exact
GLX_STERE0	False	Exact
GLX_AUX_BUFFERS	0	Smaller
GLX_RED_SIZE	0	Larger
GLX_GREEN_SIZE	0	Larger
GLX_BLUE_SIZE	0	Larger
GLX_ALPHA_SIZE	0	Larger
GLX_DEPTH_SIZE	0	Larger
GLX_STENCIL_SIZE	0	Smaller
GLX_ACCUM_RED_SIZE	0	Larger
GLX_ACCUM_GREEN_SIZE	0	Larger
GLX_ACCUM_BLUE_SIZE	0	Larger
GLX_ACCUM_ALPHA_SIZE	0	Larger

Table 3.8: Defaults and selection criteria used by **glXChooseVisual**.

If GLX_FBCONFIG_ID is specified in *attrib_list*, then it is ignored (however, if present, it must still be followed by an attribute value).

If an attribute is not specified in *attrib_list*, then the default value is used. See Table 3.8 for a list of defaults.

Default specifications are superseded by the attributes included in *attrib_list*. Integer attributes are immediately followed by the corresponding desired value. Boolean attributes appearing in *attrib_list* have an implicit True value; such attributes are *never* followed by an explicit True or False value. The list is terminated with None.

To free the data returned, use **XFree**.

NULL is returned if an undefined GLX attribute is encountered.

3.4.2 Off Screen Rendering

A GLXPixmap can be created using by calling

GLXPixmap glXCreateGLXPixmap(Display *dpy, XVisualInfo *visual, Pixmap pixmap);

Calling glXCreateGLXPixmap(dpy, visual, pixmap) is equivalent to calling glXCreatePixmap(dpy, config, pixmap, NULL) where config is the GLXFBConfig identified by the GLX_FBCONFIG_ID attribute of visual. Before calling glXCreateGLXPixmap, clients must first create an X Pixmap of the depth specified by visual. The GLXFBConfig identified by the GLX_FBCONFIG_ID attribute of visual is associated with the resulting pixmap. Any compatible GLX rendering context can be used to render into this offscreen area.

If the depth of *pixmap* does not match the depth value reported by core X11 for *visual*, or if *pixmap* was not created with respect to the same screen as *visual*, then a BadMatch error is generated. If *visual* is not valid (e.g., if GLX does not support it), then a BadValue error is generated. If *pixmap* is not a valid pixmap id, then a BadPixmap error is generated. Finally, if the server cannot allocate the new GLX pixmap, a BadAlloc error is generated.

A GLXPixmap created by glXCreateGLXPixmap can be destroyed by calling

void glXDestroyGLXPixmap(Display *dpy, GLXPixmap pixmap);

This function is equivalent to glXDestroyPixmap; however, GLXPixmaps created by calls other than glXCreateGLXPixmap should not be passed to glXDestroyGLXPixmap.

3.5 Rendering Contexts

An OpenGL rendering context may be created by calling

```
GLXContext glXCreateContext(Display *dpy,
XVisualInfo *visual, GLXContext share_list, Bool
direct);
```

Calling glXCreateContext(dpy, visual, share_list, direct) is equivalent to calling glXCreateNewContext(dpy, config, render_type, share_list, direct) where config is the GLXFBConfig identified by the GLX_FBCONFIG_ID attribute of visual. If visual's GLX_RGBA attribute is True then render_type is taken as GLX_RGBA_TYPE, otherwise GLX_COLOR_INDEX_TYPE. The GLXFBConfig identified by the GLX_FBCONFIG_ID attribute of visual is associated with the resulting context.

glXCreateContext can generate the following errors: GLXBadContext if *share_list* is neither zero nor a valid GLX rendering context; BadValue if *visual* is not a valid X Visual or if GLX does not support it; BadMatch if *share_list* defines an address space that cannot be shared with the newly created context or if *share_list* was created on a different screen than the one referenced by *visual*; BadAlloc if the server does not have enough resources to allocate the new context.

To make a context current, call

Bool glXMakeCurrent(Display *dpy, GLXDrawable draw, GLXContext ctx);

Calling glXMakeCurrent(dpy, draw, ctx) is equivalent to calling glX-MakeContextCurrent(dpy, draw, draw, ctx). Note that draw will be used for both the draw and read drawable.

If ctx and draw are not compatible then a BadMatch error will be generated. Some implementations may enforce a stricter rule and generate a BadMatch error if ctx and draw were not created with the same XVisualInfo.

If ctx is current to some other thread, then glXMakeCurrent will generate a BadAccess error. GLXBadContextState is generated if there is a current rendering context and its render mode is either GL_FEEDBACK or GL_SELECT. If ctx is not a valid GLX rendering context, GLXBadContext is generated. If draw is not a valid GLXPixmap or a valid Window, a GLXBadDrawable error is generated. If the previous context of the calling thread has unflushed commands, and the previous drawable is a window that is no longer valid, GLXBadCurrentWindow is generated. Finally, note that

3.5. RENDERING CONTEXTS

the ancillary buffers for *draw* need not be allocated until they are needed. A BadAlloc error will be generated if the server does not have enough resources to allocate the buffers.

To release the current context without assigning a new one, use NULL for ctx and None for draw. If ctx is NULL and draw is not None, or if draw is None and ctx is not NULL, then a BadMatch error will be generated.

Chapter 4

Encoding on the X Byte Stream

In the remote rendering case, the overhead associated with interpreting the GLX extension requests must be minimized. For this reason, all commands have been broken up into two categories: OpenGL and GLX commands that are each implemented as a single X extension request and OpenGL rendering requests that are batched within a GLXRender request.

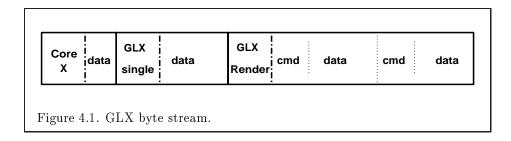
4.1 Requests that hold a single extension request

Each of the commands from $\langle glx.h \rangle$ (that is, the glX^* commands) is encoded by a separate X extension request. In addition, there is a separate X extension request for each of the OpenGL commands that cannot be put into a display list. That list consists of all the $glGet^*$ commands plus

glAreTexturesResident glDeleteLists glDeleteTextures glEndList glFeedbackBuffer glFinish glFlush glGenLists glGenTextures glIsEnabled glIsList

42

4.2. REQUEST THAT HOLDS MULTIPLE OPENGL COMMANDS 43



glIsTexture glNewList glPixelStoref glPixelStorei glReadPixels glRenderMode glSelectBuffer

The two **PixelStore** commands (**glPixelStorei** and **glPixelStoref**) are exceptions. These commands are issued to the server only to allow it to set its error state appropriately. Pixel storage state is maintained entirely on the client side. When pixel data is transmitted to the server (by **glDrawPixels**, for example), the pixel storage information that describes it is transmitted as part of the same protocol request. Implementations may not change this behavior, because such changes would cause shared contexts to behave incorrectly.

4.2 Request that holds multiple OpenGL commands

The remaining OpenGL commands are those that may be put into display lists. Multiple occurrences of these commands are grouped together into a single X extension request (**GLXRender**). This is diagrammed in Figure 4.1.

The grouping minimizes dispatching within the X server. The library packs as many OpenGL commands as possible into a single X request (without exceeding the maximum size limit). No OpenGL command may be split across multiple **GLXRender** requests.

For OpenGL commands whose encoding is longer than the maximum

X request size, a series of **GLXRenderLarge** commands are issued. The structure of the OpenGL command within **GLXRenderLarge** is the same as for **GLXRender**.

Note that it is legal to have a **glBegin** in one request, followed by **glVertex** commands, and eventually the matching **glEnd** in a subsequent request. A command is not the same as an OpenGL primitive.

4.3 Wire representations and byte swapping

Unsigned and signed integers are represented as they are represented in the core X protocol. Single and double precision floating point numbers are sent and received in IEEE floating point format. The X byte stream and network specifications make it impossible for the client to assure that double precision floating point numbers will be naturally aligned within the transport buffers of the server. For those architectures that require it, the server or client must copy those floating point numbers to a properly aligned buffer before using them.

Byte swapping on the encapsulated OpenGL byte stream is performed by the server using the same rule as the core X protocol. Single precision floating point values are swapped in the same way that 32-bit integers are swapped. Double precision floating point values are swapped across all 8 bytes.

4.4 Sequentiality

There are two sequences of commands: the X stream, and the OpenGL stream. In general these two streams are independent: Although the commands in each stream will be processed in sequence, there is no guarantee that commands in the separate streams will be processed in the order in which they were issued by the calling thread.

An exception to this rule arises when a single command appears in *both* streams. This forces the two streams to rendezvous.

Because the processing of the two streams may take place at different rates, and some operations may depend on the results of commands in a different stream, we distinguish between commands assigned to each of the X and OpenGL streams.

The following commands are processed on the client side and therefore do not exist in either the X or the OpenGL stream: glXGetClientString glXGetCurrentContext glXGetCurrentDisplay glXGetCurrentDrawable glXGetCurrentReadDrawable glXGetConfig glXGetFBConfigAttrib glXGetFBConfigs glXGetSelectedEvent glXGetVisualFromFBConfig

The following commands are in the X stream and obey the sequentiality guarantees for X requests:

glXChooseFBConfig glXChooseVisual glXCreateContext glXCreateGLXPixmap glXCreateNewContext glXCreatePbuffer glXCreatePixmap **glXCreateWindow** glXDestroyContext glXDestroyGLXPixmap glXDestroyPbuffer glXDestroyPixmap glXDestroyWindow glXMakeContextCurrentglXMakeCurrent glXIsDirect glXQueryContext glXQueryDrawable glXQueryExtension glXQueryExtensionsString glXQueryServerString glXQueryVersion glXSelectEvent glXWaitGL glXSwapBuffers (see below) glXCopyContext (see below)

glXSwapBuffers is in the X stream if and only if the display and drawable are not those belonging to the calling thread's current context; otherwise it is in the OpenGL stream. glXCopyContext is in the X stream alone if and only if its source context differs from the calling thread's current context; otherwise it is in both streams.

Commands in the OpenGL stream, which obey the sequentiality guarantees for OpenGL requests are:

glXWaitX glXSwapBuffers (see below) All OpenGL Commands

glXSwapBuffers is in the OpenGL stream if and only if the display and drawable are those belonging to the calling thread's current context; otherwise it is in the X stream.

Commands in both streams, which force a rendezvous, are:

glXCopyContext (see below) glXUseXFont

glXCopyContext is in both streams if and only if the source context is the same as the current context of the calling thread; otherwise it is in the X stream only.

Chapter 5

Extending OpenGL

OpenGL implementors may extend OpenGL by adding new OpenGL commands or additional enumerated values for existing OpenGL commands. When a new vendor-specific command is added, GLX protocol must also be defined. If the new command is one that cannot be added to a display list, then protocol for a new glXVendorPrivate or glXVendorPrivateWith-Reply request is required; otherwise protocol for a new rendering command that can be sent to the X Server as part of a glXRender or glXRender-Large request is required.

The OpenGL Architectural Review Board maintains a registry of vendorspecific enumerated values; opcodes for vendor private requests, vendor private with reply requests, and OpenGL rendering commands; and vendorspecific error codes and event codes.

New names for OpenGL functions and enumerated types must clearly indicate whether some particular feature is in the core OpenGL or is vendor specific. To make a vendor-specific name, append a company identifier (in upper case) and any additional vendor-specific tags (e.g. machine names). For instance, SGI might add new commands and manifest constants of the form glNewCommandSGI and GL_NEW_DEFINITION_SGI. If two or more licensees agree in good faith to implement the same extension, and to make the specification of that extension publicly available, the procedures and tokens that are defined by the extension can be suffixed by EXT.

Implementors may also extend GLX. As with OpenGL, the new names must indicate whether or not the feature is vendor-specific. (e.g., SGI might add new GLX commands and constants of the form glXNewCommandSGI and GLX_NEW_DEFINITION_SGI). When a new GLX command is added, protocol for a new glXVendorPrivate or glXVendorPrivateWithReply request is required.

Chapter 6

GLX Versions

Each version of GLX supports all versions of OpenGL up to the version shown in Table 6.1 corresponding to the given GLX version.

6.1 New Commands in GLX Version 1.1

The following GLX commands were added in GLX Version 1.1:

glXQueryExtensionsString glXGetClientString glXQueryServerString

6.2 New Commands in GLX Version 1.2

The following GLX commands were added in GLX Version 1.2:

GLX Version	Highest OpenGL	
	Version Supported	
GLX 1.0	OpenGL 1.0	
GLX 1.1	OpenGL 1.0	
GLX 1.2	OpenGL 1.1	
GLX 1.3	OpenGL 1.2	

Table 6.1: Relationship of OpenGL and GLX versions.

glXGetCurrentDisplay

6.3 New Commands in GLX Version 1.3

The following GLX commands were added in GLX Version 1.3:

glXChooseFBConfig glXGetFBConfigAttribglXGetVisualFromFBConfig glXCreateWindow glXDestroyWindow glXCreatePixmap glXDestroyPixmap glXCreatePbufferglXDestroyPbuffer glXQueryDrawableglXCreateNewContextglXMakeContextCurrentglXGetCurrentReadDrawableglXQueryContext glXSelectEventglXGetSelectedEvent

Chapter 7

Glossary

- Address Space the set of objects or memory locations accessible through a single name space. In other words, it is a data region that one or more processes may share through pointers.
- **Client** an X client. An application communicates to a server by some path. The application program is referred to as a client of the window system server. To the server, the client is the communication path itself. A program with multiple connections is viewed as multiple clients to the server. The resource lifetimes are controlled by the connection lifetimes, not the application program lifetimes.
- **Compatible** an OpenGL rendering context is compatible with (may be used to render into) a GLXDrawable if they meet the constraints specified in section 2.1.
- **Connection** a bidirectional byte stream that carries the X (and GLX) protocol between the client and the server. A client typically has only one connection to a server.
- (Rendering) Context a OpenGL rendering context. This is a virtual OpenGL machine. All OpenGL rendering is done with respect to a context. The state maintained by one rendering context is not affected by another except in case of shared display lists and textures.
- **GLXContext** a handle to a rendering context. Rendering contexts consist of client side state and server side state.
- Similar a potential correspondence among GLXDrawables and rendering contexts. Windows and GLXPixmaps are similar to a rendering context

are similar if, and only if, they have been created with respect to the same VisualID and root window.

Thread one of a group of processes all sharing the same address space. Typically, each thread will have its own program counter and stack pointer, but the text and data spaces are visible to each of the threads. A thread that is the only member of its group is equivalent to a process.

Index of GLX Commands

BadAccess, 27, 29, 40 BadAlloc, 21, 23, 24, 26, 27, 39–41 BadFont, 35 BadMatch, 21, 23, 24, 26–29, 39–41 BadPixmap, 23, 39 BadValue, 26, 39, 40 BadWindow, 21

GL_ALL_ATTRIB_BITS, 29 GL_BACK, 28 GL_DRAW_BUFFER, 28 GL_FEEDBACK, 9, 27, 40 GL_NEW_DEFINITION_SGI, 47 GL_READ_BUFFER,27,28 GL_SELECT, 9, 27, 40 GL_TEXTURE_1D, 6 GL_TEXTURE_2D, 6 GL_TEXTURE_3D, 6 glAreTexturesResident, 42 glBegin, 9, 10, 44 glBindTexture, 6 glBitmap, 34 glCopyColorTable, 28 glCopyPixels, 27, 28 glDeleteLists, 6, 42 glDeleteTextures, 42 glDrawBuffer, 28 glDrawPixels, 43 glEnd, 9, 10, 44 glEndList, 6, 42 glFeedbackBuffer, 42 glFinish, 8, 33, 42 glFlush, 3, 29, 34, 35, 42 glGenLists, 42 glGenTextures, 42 glGet, 28

 $glGet^*, 5, 42$ glIsEnabled, 42 glIsList, 42 glIsTexture, 43 glListBase, 6 glNewCommandSGI, 47 glNewList, 6, 43 glPixelStoref, 43 glPixelStorei, 43 glPopAttrib, 28, 29 glPushAttrib, 28, 29 glReadBuffer, 28 glReadPixels, 24, 27, 28, 43 glRenderMode, 9, 43 glScissor, 28 glSelectBuffer, 43 glVertex, 44 glViewport, 28 glX*, 42 GLX_ACCUM_ALPHA_SIZE, 13, 19, 20,36,38 GLX_ACCUM_BLUE_SIZE, 13, 19, 20,36,38 GLX_ACCUM_BUFFER_BIT, 32 GLX_ACCUM_GREEN_SIZE, 13, 19, 20,36,38 GLX_ACCUM_RED_SIZE,13,19,20, 36, 38 GLX_ALPHA_SIZE, 12, 13, 18, 19, 36, 38 GLX_AUX_BUFFERS, 13, 18, 19, 36, 38 GLX_AUX_BUFFERS_BIT, 32 GLX_BACK_LEFT_BUFFER_BIT, 32

53

GLX_BACK_RIGHT_BUFFER_BIT, 32GLX_BAD_ATTRIBUTE, 20, 30, 37 GLX_BAD_SCREEN, 37 GLX_BAD_VALUE, 37 GLX_BAD_VISUAL, 37 GLX_BLUE_SIZE, 12, 13, 18, 19, 36, 38 GLX_BUFFER_SIZE, 12, 13, 18, 19, 36, 38GLX_COLOR_INDEX_BIT, 12, 15 GLX_COLOR_INDEX_TYPE,25,40 GLX_CONFIG_CAVEAT,13,15,18, 19GLX_DAMAGED, 31, 33 GLX_DEPTH_BUFFER_BIT, 32 GLX_DEPTH_SIZE,13,19,20,36, 38 GLX_DIRECT_COLOR,14, 20 GLX_DONT_CARE, 17-20 GLX_DOUBLE_BUFFER, 18 GLX_DOUBLEBUFFER, 13, 19, 36, 38 GLX_DRAWABLE_TYPE, 13--15, 18 - 21GLX_EXTENSIONS, 11 GLX_FBCONFIG_ID, 13, 18, 19, 25, 30, 36, 39, 40 GLX_FRONT_LEFT_BUFFER_BIT, 32GLX_FRONT_RIGHT_BUFFER_ BIT, 32 GLX_GRAY_SCALE, 14, 20 GLX_GREEN_SIZE, 12, 13, 18, 19, 36, 38GLX_HEIGHT, 25 GLX_LARGEST_PBUFFER,23, 25 GLX_LEVEL, 13, 17, 19, 36, 38 GLX_MAX_PBUFFER_HEIGHT,13, 16, 18 GLX_MAX_PBUFFER_PIXELS,13, 16, 18 GLX_MAX_PBUFFER_WIDTH,13, 16, 18 GLX_NEW_DEFINITION_SGI, 47

GLX_NO_EXTENSION, 37

GLX_NON_CONFORMANT_ CONFIG, 15, 18 GLX_NONE,14--16,18, 19 GLX_PBUFFER, 31 GLX_PBUFFER_BIT, 14 GLX_PBUFFER_CLOBBER_ MASK, 31 GLX_PBUFFER_HEIGHT, 23 GLX_PBUFFER_WIDTH, 23 GLX_PbufferClobber, 10 GLX_PIXMAP_BIT, 14 GLX_PRESERVED_CONTENTS, 23 - 25GLX_PSEUDO_COLOR,14, 20 GLX_RED_SIZE, 12, 13, 16--19, 36, 38 GLX_RENDER_TYPE, 12, 13, 15, 19, 30 GLX_RGBA, 36--38, 40 GLX_RGBA_BIT, 12, 15, 19 GLX_RGBA_TYPE,25,40 GLX_SAVED, 31-33 GLX_SCREEN, 30 GLX_SLOW_CONFIG, 15, 18 GLX_STATIC_COLOR,14, 20 GLX_STATIC_GRAY,14, 20 GLX_STENCIL_BITS, 20 GLX_STENCIL_BUFFER_BIT, 32 GLX_STENCIL_SIZE, 13, 19, 36, 38 GLX_STEREO, 13, 17, 19, 36, 38 GLX_TRANSPARENT_ALPHA_ VALUE, 13, 16, 18, 19 GLX_TRANSPARENT_BLUE_ VALUE, 13, 16, 18, 19 GLX_TRANSPARENT_GREEN_ VALUE, 13, 16, 18, 19 GLX_TRANSPARENT_INDEX, 16 GLX_TRANSPARENT_INDEX_ VALUE, 13, 16, 18, 19 GLX_TRANSPARENT_RED_ VALUE, 13, 16, 18, 19 GLX_TRANSPARENT_RGB, 16 GLX_TRANSPARENT_TYPE,13, 15, 16, 18, 19 GLX_TRUE_COLOR,14, 20 GLX_USE_GL, 36-38

INDEX

GLX_VENDOR, 11 GLX_VERSION, 11 GLX_VISUAL_ID, 13, 14, 18 GLX_WIDTH, 25 GLX_WINDOW, 31 GLX_WINDOW_BIT, 14, 15, 18-21 GLX_X_RENDERABLE,13,14, 19 GLX_X_VISUAL_TYPE,13,14, 18-20GLXBadContext, 9, 26, 27, 29, 30, 40 GLXBadContextState, 9, 27, 35, 40 GLXBadContextTag, 10 GLXBadCurrentDrawable, 9, 27, 29, 33 - 35GLXBadCurrentWindow, 9, 40 GLXBadDrawable, 9, 25, 27, 31, 34, 40GLXBadFBConfig, 9, 23, 24, 26 GLXBadLargeRequest, 10 GLXBadPbuffer, 9, 25 GLXBadPixmap, 10, 23 GLXBadRenderRequest, 10 GLXBadWindow, 10, 21, 27, 34 glXChooseFBConfig, 12, 17, 20-23, 25.45.50 glXChooseVisual, 35, 37, 38, 45 GLXContext, 12 glXCopyContext, 28, 29, 46 glXCreateContext, 40, 45 glXCreateGLXPixmap, 39, 45 glXCreateNewContext, 25, 26, 40, 45, 50glXCreatePbuffer, 16, 23, 24, 45, 50 glXCreatePixmap, 3, 22, 39, 45, 50 glXCreateWindow, 21, 45, 50 glXDestroyContext, 26, 45 glXDestroyGLXPixmap, 39, 45 glXDestroyPbuffer, 25, 45, 50 glXDestroyPixmap, 23, 39, 45, 50 glXDestroyWindow, 21, 45, 50 GLXDrawable, 2, 3, 12, 22, 25, 27, 31, 32, 35, 51 GLXFBConfig, 2, 3, 9, 12-26, 29, 30, 34-37, 39, 40 GLXFBConfigs, 17, 18

glXGet*, 30 glXGetClientString, 11, 12, 45, 49 glXGetConfig, 35, 37, 45 glXGetCurrentContext, 30, 45 glXGetCurrentDisplay, 30, 45, 50 glXGetCurrentDrawable, 30, 45 glXGetCurrentReadDrawable, 30, 45, 50glXGetFBConfigAttrib, 20, 45, 50 glXGetFBConfigs, 12, 16, 45 glXGetSelectedEvent, 31, 45, 50 glXGetVisualFromFBConfig, 20, 21, 45, 50 glXIsDirect, 26, 45 glXMakeContextCurrent, 26-28, 40, 45, 50 glXMakeCurrent, 9, 40, 45 glXNewCommandSGI, 47 GLXPbuffer, 2, 3, 9, 12, 14, 16, 21-25, 27, 31, 35 GLXPbufferClobberEvent, 32 GLXPixmap, 2, 3, 12, 14, 21-23, 25, 27, 34, 35, 39, 40, 51 glXQuery^{*}, 30 glXQueryContext, 30, 45, 50 glXQueryDrawable, 23, 25, 45, 50 glXQueryExtension, 10, 45 glXQueryExtensionsString, 11, 45, 49 glXQueryServerString, 12, 45, 49 glXQueryVersion, 11, 45 GLXRender, 42 glXSelectEvent, 24, 31, 45, 50 glXSwapBuffers, 22, 24, 34, 45, 46 GLXUnsupportedPrivateRequest, 10 glXUseXFont, 34, 35, 46 glXWaitGL, 8, 33, 45 glXWaitX, 8, 33, 46 GLXWindow, 2, 3, 10, 12, 21, 25, 27, 28, 31 None, 17, 21-23, 28, 30, 39, 41 PixelStore, 43

Screen, 35

INDEX

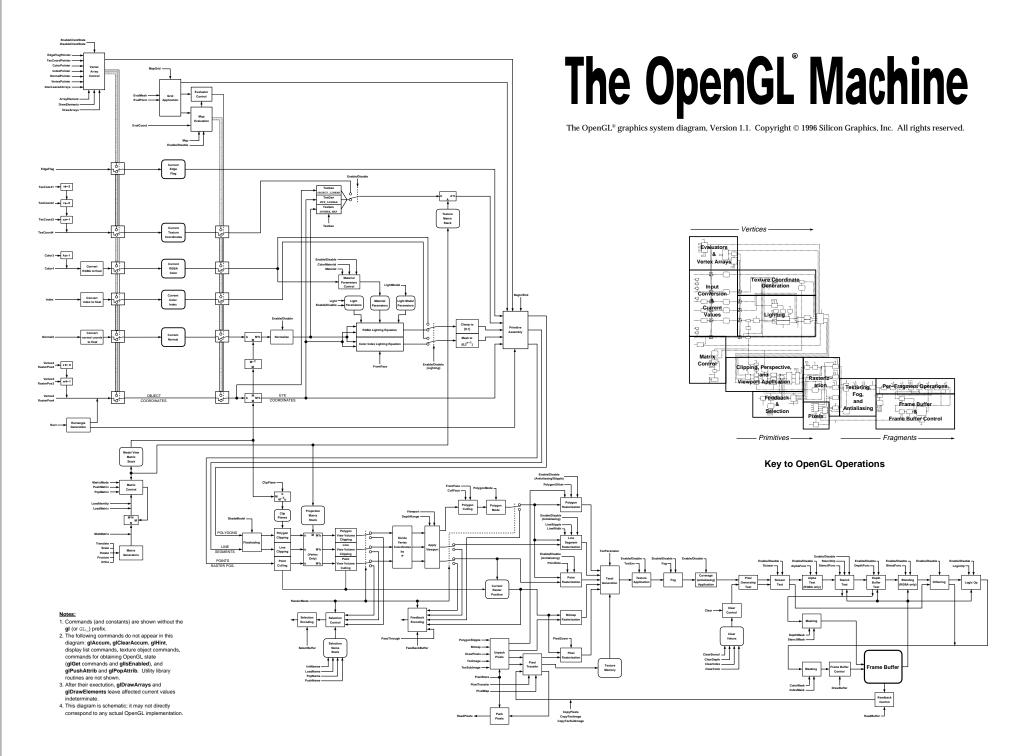
Success, 20, 30, 37

Visual, 3, 12, 14, 20, 22, 35–37, 40 VisualID, 35

Window, 2, 3, 9, 21, 27, 28, 34, 40 Windows, 35

XFree, 20, 39 XFreePixmap, 23 XSync, 8, 24, 33 XVisualInfo, 35

56



SIGGRAPH '97

Course 24: OpenGL and Window System Integration

OpenGL Performance Optimization

Contents

- <u>1. Hardware vs. Software</u>
- <u>2. Application Organization</u>
 - <u>2.1 High Level Organization</u>
 - <u>2.2 Low Level Organization</u>
- <u>3. OpenGL Optimization</u>
 - <u>3.1 Traversal</u>
 - <u>3.2 Transformation</u>
 - <u>3.3 Rasterization</u>
 - <u>3.4 Texturing</u>
 - <u>3.5 Clearing</u>
 - <u>3.6 Miscellaneous</u>
 - <u>3.7 Window System Integration</u>
 - <u>3.8 Mesa-specific</u>
- <u>4. Evaluation and tuning</u>
 - <u>4.1 Pipeline tuning</u>
 - <u>4.2 Double buffering</u>
 - <u>4.3 Test on several implementations</u>

1. Hardware vs. Software

OpenGL may be implemented by any combination of hardware and software. At the high-end, hardware may implement virtually all of OpenGL while at the low-end, OpenGL may be implemented entirely in software. In between are combination software/hardware implementations. More money buys more hardware and better performance.

Intro-level workstation hardware and the recent PC 3-D hardware typically implement point, line, and polygon rasterization in hardware but implement floating point transformations, lighting, and clipping in software. This is a good strategy since the bottleneck in 3-D rendering is usually rasterization and modern CPU's have sufficient floating point performance to handle the transformation stage.

OpenGL developers must remember that their application may be used on a wide variety of OpenGL implementations. Therefore one should consider using all possible optimizations, even those which have little return on the development system, since other systems may benefit greatly.

From this point of view it may seem wise to develop your application on a low-end system. There is a pitfall however; some operations which are cheep in software may be expensive in hardware. The moral is: test your application on a variety of systems to be sure the performance is dependable.

2. Application Organization

At first glance it may seem that the performance of interactive OpenGL applications is dominated by the performance of OpenGL itself. This may be true in some circumstances but be aware that the organization of the application is also significant.

2.1 High Level Organization

Multiprocessing

Some graphical applications have a substantial computational component other than 3-D rendering. Virtual reality applications must compute object interactions and collisions. Scientific visualization programs must compute analysis functions and graphical representations of data.

One should consider multiprocessing in these situations. By assigning rendering and computation to different threads they may be executed in parallel on multiprocessor computers.

For many applications, supporting multiprocessing is just a matter of partitioning the render and compute operations into separate threads which share common data structures and coordinate with synchronization primitives.

SGI's Performer is an example of a high level toolkit designed for this purpose.

Image quality vs. performance

In general, one wants high-speed animation and high-quality images in an OpenGL application. If you can't have both at once a reasonable compromise may be to render at low complexity during animation and high complexity for static images.

Complexity may refer to the geometric or rendering attributes of a database. Here are a few examples.

- During interactive rotation (i.e. mouse button held down) render a reduced-polygon model. When drawing a static image draw the full polygon model.
- During animation, disable dithering, smooth shading, and/or texturing. Enable them for the static image.
- If texturing is required, use GL_NEAREST sampling and glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST).
- During animation, disable antialiasing. Enable antialiasing for the static image.
- Use coarser NURBS/evaluator tesselation during animation. Use glPolygonMode(GL_FRONT_AND_BACK, GL_LINE) to inspect tesselation granularity and reduce if possible.

Level of detail management and culling

Objects which are distant from the viewer may be rendered with a reduced complexity model. This strategy reduces the demands on all stages of the graphics pipeline. Toolkits such as Inventor and Performer support this feature automatically.

Objects which are entirely outside of the field of view may be culled. This type of high level cull testing can be done efficiently with bounding boxes or spheres and have a major impact on performance. Again, toolkits such as Inventor and Performer have this feature.

2.2 Low Level Organization

The objects which are rendered with OpenGL have to be stored in some sort of data structure. Some data structures are more efficient than others with respect to how quickly they can be rendered.

Basically, one wants data structures which can be traversed quickly and passed to the graphics library in an efficient manner. For example, suppose we need to render a triangle strip. The data structure which stores the list of vertices may be implemented with a linked list or an array. Clearly the array can be traversed more quickly than a linked list. The way in which a vertex is stored in the data structure is also significant. High performance hardware can process vertexes specified by a pointer more quickly than those specified by three separate parameters.

An Example

Suppose we're writing an application which involves drawing a road map. One of the components of the database is a list of cities specified with a latitude, longitude and name. The data structure describing a city may be:

```
struct city {
```

};

```
float latitute, longitude; /* city location */
char *name; /* city's name */
int large_flag; /* 0 = small, 1 = large */
```

A list of cities may be stored as an array of city structs.

Our first attempt at rendering this information may be:

```
void draw_cities( int n, struct city citylist[] )
{
   int i;
   for (i=0; i < n; i++) {
      if (citylist[i].large_flag) {
          glPointSize( 4.0 );
      }
      else {
         glPointSize( 2.0 );
      }
      glBegin( GL_POINTS );
     glVertex2f( citylist[i].longitude, citylist[i].latitude );
     glEnd();
     glRasterPos2f( citylist[i].longitude, citylist[i].latitude );
      glCallLists( strlen(citylist[i].name),
                     GL_BYTE,
                     citylist[i].name );
   }
}
```

This is a poor implementation for a number of reasons:

- glPointSize is called for every loop iteration.
- only one point is drawn between glBegin and glEnd
- the vertices aren't being specified in the most efficient manner

Here's a better implementation:

```
void draw_cities( int n, struct city citylist[] )
ł
   int i;
   /* draw small dots first */
  glPointSize( 2.0 );
  glBegin( GL_POINTS );
   for (i=0; i < n ;i++) {
      if (citylist[i].large_flag==0) {
         glVertex2f( citylist[i].longitude, citylist[i].latitude );
      }
   }
  glEnd();
   /* draw large dots second */
  glPointSize( 4.0 );
  glBegin( GL_POINTS );
   for (i=0; i < n ;i++) {
      if (citylist[i].large_flag==1) {
         glVertex2f( citylist[i].longitude, citylist[i].latitude );
      }
   }
  glEnd();
   /* draw city labels third */
   for (i=0; i < n ;i++) {</pre>
     glRasterPos2f( citylist[i].longitude, citylist[i].latitude );
      glCallLists( strlen(citylist[i].name),
                     GL_BYTE,
                     citylist[i].name );
   }
}
```

In this implementation we're only calling glPointSize twice and we're maximizing the number of vertices specified between glBegin and glEnd.

We can still do better, however. If we redesign the data structures used to represent the city information we can improve the efficiency of drawing the city points. For example:

Now cities of different sizes are stored in separate lists. Position are stored sequentially in a dynamically allocated array. By reorganizing the data structures we've eliminated the need for a conditional inside the glBegin/glEnd loops. Also, we can render a list of cities using the GL_EXT_vertex_array extension if available, or at least use a more efficient version of glVertex and glRasterPos.

```
/* indicates if server can do GL_EXT_vertex_array: */
GLboolean varray_available;
void draw_cities( struct city_list *list )
{
    int i;
    GLboolean use_begin_end;
    /* draw the points */
    glPointSize( list->size );
#ifdef GL_EXT_vertex_array
    if (varray_available) {
         glVertexPointerEXT( 2, GL_FLOAT, 0, list->num_cities, list->position );
         glDrawArraysEXT( GL_POINTS, 0, list->num_cities );
         use_begin_end = GL_FALSE;
    }
    else
#else
    {
         use_begin_end = GL_TRUE;
    }
#endif
    if (use_begin_end) {
         glBegin(GL_POINTS);
         for (i=0; i < list->num_cities; i++) {
             glVertex2fv( &position[i*2] );
         }
         glEnd();
    }
    /* draw city labels */
    for (i=0; i < list->num_cities ;i++) {
         glRasterPos2fv( list->position[i*2]
                                              );
         glCallLists( strlen(list->name[i]),
                           GL_BYTE, list->name[i] );
    }
}
```

As this example shows, it's better to know something about efficient rendering techniques before designing the data structures. In many cases one has to find a compromize between data structures optimized for rendering and those optimized for clarity and convenience.

In the following sections the techniques for maximizing performance, as seen above, are explained.

3. OpenGL Optimization

There are many possibilities to improving OpenGL performance. The impact of any single optimization can vary a great deal depending on the OpenGL implementation. Interestingly, items which have a large impact on software renderers may have no effect on hardware renderers, *and vice versa*! For example, smooth shading can be expensive in software but free in hardware While <code>glGet</code> can be cheap in software but expensive in hardware.

After each of the following techniques look for a bracketed list of symbols which relates the significance of the optimization to your OpenGL system:

- H beneficial for high-end hardware
- L beneficial for low-end hardware
- S beneficial for software implementations
- all probably beneficial for all implementations

3.1 Traversal

Traversal is the sending of data to the graphics system. Specifically, we want to minimize the time taken to specify primitives to OpenGL.

Use connected primitives

Connected primitives such as *GL_LINES*, *GL_LINE_LOOP*, *GL_TRIANGLE_STRIP*, *GL_TRIANGLE_FAN*, and *GL_QUAD_STRIP* require fewer vertices to describe an object than individual line, triangle, or polygon primitives. This reduces data transfer and transformation workload. [all]

Use the vertex array extension

On some architectures function calls are somewhat expensive so replacing many glVertex/glColor/glNormal calls with the vertex array mechanism may be very beneficial. [all]

Store vertex data in consecutive memory locations

When maximum performance is needed on high-end systems it's good to store vertex data in contiguous memory to maximize through put of data from host memory to graphics subsystem. [H,L]

Use the vector versions of glVertex, glColor, glNormal and glTexCoord

The glVertex, glColor, etc. functions which take a pointer to their arguments such as glVertex3fv(v) may be much faster than those which take individual arguments such as glVertex3f(x,y,z) on systems with DMA-driven graphics hardware. [H,L]

Reduce quantity of primitives

Be careful not to render primitives which are over-tesselated. Experiment with the GLU primitives, for example, to determine the best compromise of image quality vs. tesselation level. Textured objects in particular may still be rendered effectively with low geometric complexity. [all]

Display lists

Use display lists to encapsulate frequently drawn objects. Display list data may be stored in the graphics subsystem rather than host memory thereby eliminating host-to-graphics data movement. Display lists are also very beneficial when rendering remotely. [all]

Don't specify unneeded per-vertex information

If lighting is disabled don't call glNormal. If texturing is disabled don't call glTexCoord, etc.

Minimize code between glBegin/glEnd

For maximum performance on high-end systems it's extremely important to send vertex data to the graphics system as fast as possible. Avoid extraneous code between glBegin/glEnd.

Example:

glBegin(GL_TRIANGLE_STRIP); for (i=0; i < n; i++) { if (lighting) {

```
glNormal3fv( norm[i] );
}
glVertex3fv( vert[i] );
}
glEnd();
```

This is a very bad construct. The following is much better:

```
if (lighting) {
    glBegin( GL_TRIANGLE_STRIP );
    for (i=0; i < n ; i++) {
        glNormal3fv( norm[i] );
        glVertex3fv( vert[i] );
    }
    glEnd();
}
else {
    glBegin( GL_TRIANGLE_STRIP );
    for (i=0; i < n ; i++) {
            glVertex3fv( vert[i] );
        }
        glEnd();
}
</pre>
```

Also consider manually unrolling important rendering loops to maximize the function call rate.

3.2 Transformation

Transformation includes the transformation of vertices from *glVertex* to window coordinates, clipping and lighting.

Lighting

- Avoid using positional lights, i.e. light positions should be of the form (x,y,z,0) [L,S]
- Avoid using spotlights. [all]
- Avoid using two-sided lighting. [all]
- Avoid using negative material and light color coefficients [S]
- Avoid using the local viewer lighting model. [L,S]
- Avoid frequent changes to the GL_SHININESS material parameter. [L,S]
- Some OpenGL implementations are optimized for the case of a single light source.
- Consider pre-lighting complex objects before rendering, ala radiosity. You can get the effect of lighting by specifying vertex colors instead of vertex normals. [S]

Two sided lighting

If you want both the front and back of polygons shaded the same try using two light sources instead of two-sided lighting. Position the two light sources on opposite sides of your object. That way, a polygon will always be lit correctly whether it's back or front facing. [L,S]

Disable normal vector normalization when not needed

glEnable/Disable(GL_NORMALIZE) controls whether normal vectors are scaled to unit length before lighting. If you do not use glScale you may be able to disable normalization without ill effects. Normalization is disabled by default. [L,S]

Use connected primitives

glRect usage

If you have to draw many rectangles consider using glBegin(GL_QUADS) ... glEnd() instead. [all]

3.3 Rasterization

Rasterization is the process of generating the pixels which represent points, lines, polygons, bitmaps and the writing of

those pixels to the frame buffer. Rasterization is often the bottleneck in software implementations of OpenGL.

Disable smooth shading when not needed

Smooth shading is enabled by default. Flat shading doesn't require interpolation of the four color components and is usually faster than smooth shading in software implementations. Hardware may perform flat and smooth-shaded rendering at the same rate though there's at least one case in which smooth shading is faster than flat shading (E&S Freedom). [S]

Disable depth testing when not needed

Background objects, for example, can be drawn without depth testing if they're drawn first. Foreground objects can be drawn without depth testing if they're drawn last. [L,S]

Disable dithering when not needed

This is easy to forget when developing on a high-end machine. Disabling dithering can make a big difference in software implementations of OpenGL on lower-end machines with 8 or 12-bit color buffers. Dithering is enabled by default. [S]

Use back-face culling whenever possible.

If you're drawing closed polyhedra or other objects for which back facing polygons aren't visible there's probably no point in drawing those polygons. [all]

The GL_SGI_cull_vertex extension

SGI's Cosmo GL supports a new culling extension which looks at vertex normals to try to improve the speed of culling.

Avoid extra fragment operations

Stenciling, blending, stippling, alpha testing and logic ops can all take extra time during rasterization. Be sure to disable the operations which aren't needed. [all]

Reduce the window size or screen resolution

A simple way to reduce rasterization time is to reduce the number of pixels drawn. If a smaller window or reduced display resolution are acceptable it's an easy way to improve rasterization speed. [L,S]

3.4 Texturing

Texture mapping is usually an expensive operation in both hardware and software. Only high-end graphics hardware can offer free to low-cost texturing. In any case there are several ways to maximize texture mapping performance.

Use efficient image formats

The $_{GL_UNSIGNED_BYTE}$ component format is typically the fastest for specifying texture images. Experiment with the internal texture formats offered by the $_{GL_EXT_texture}$ extension. Some formats are faster than others on some systems (16-bit texels on the Reality Engine, for example). [all]

Encapsulate texture maps in texture objects or display lists

This is especially important if you use several texture maps. By putting textures into display lists or texture objects the graphics system can manage their storage and minimize data movement between the client and graphics subsystem. [all]

Use smaller texture maps

Smaller images can be moved from host to texture memory faster than large images. More small texture can be stored simultaneously in texture memory, reducing texture memory swapping. [all]

Use simpler sampling functions

Experiment with the minification and magnification texture filters to determine which performs best while giving acceptable results. Generally, GL_NEAREST is fastest and GL_LINEAR is second fastest. [all]

Use the same sampling function for minification and magnification

If both the minification and magnification filters are _{GL_NEAREST} or _{GL_LINEAR} then there's no reason OpenGL has to compute the *lambda* value which determines whether to use minification or magnification sampling for each fragment. Avoiding the lambda calculation can be a good performace improvement.

Use a simpler texture environment function

Some texture environment modes may be faster than others. For example, the _{GL_DECAL} or _{GL_REPLACE_EXT} functions for 3 component textures is a simple assignment of texel samples to fragments while _{GL_MODULATE} is a linear interpolation between texel samples and incoming fragments. [S,L]

Combine small textures

If you are using several small textures consider tiling them together as a larger texture and modify your texture coordinates to address the subtexture you want. This technique can eliminate texture bindings.

Use glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST)

This hint can improve the speed of texturing when perspective - correct texture coordinate interpolation isn't needed, such as when using a glOrtho() projection.

Animated textures

If you want to use an animated texture, perhaps live video textures, don't use glTexImage2D to repeatedly change the texture. Use glTexSubImage2D or glTexCopyTexSubImage2D. These functions are standard in OpenGL 1.1 and available as extensions to 1.0.

3.5 Clearing

Clearing the color, depth, stencil and accumulation buffers can be time consuming, especially when it has to be done in software. There are a few tricks which can help.

```
Use glClear carefully [all]
```

Clear all relevant color buffers with one glClear.

Wrong:

```
glClear( GL_COLOR_BUFFER_BIT );
if (stenciling) {
     glClear( GL_STENCIL_BUFFER_BIT );
}
```

Right:

```
if (stenciling) {
    glClear( GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
}
else {
    glClear( GL_COLOR_BUFFER_BIT );
}
```

Disable dithering

Disable dithering before clearing the color buffer. Visually, the difference between dithered and undithered clears is usually negligable.

Use scissoring to clear a smaller area

If you don't need to clear the whole buffer use glscissor() to restrict clearing to a smaller area. [L].

Don't clear the color buffer at all

If the scene you're drawing opaquely covers the entire window there is no reason to clear the color buffer.

Eliminate depth buffer clearing

If the scene you're drawing covers the entire window there is a trick which let's you omit the depth buffer clear. The idea is to only use half the depth buffer range for each frame and alternate between using GL_LESS and GL_GREATER as the depth test function.

Example:

int EvenFlag;

/* Call this once during initialization and whenever the window

```
* is resized.
 * /
void init_depth_buffer( void )
{
      glClearDepth( 1.0 );
      glClear( GL_DEPTH_BUFFER_BIT );
      glDepthRange( 0.0, 0.5 );
      glDepthFunc( GL_LESS );
      EvenFlag = 1;
}
/* Your drawing function */
void display_func( void )
{
      if (EvenFlag) {
          glDepthFunc( GL_LESS );
          glDepthRange( 0.0, 0.5 );
      }
      else {
          glDepthFunc( GL_GREATER );
          glDepthRange( 1.0, 0.5 );
      EvenFlag = !EvenFlag;
      /* draw your scene */
}
```

Avoid glClearDepth(d) where d!=1.0

Some software implementations may have optimized paths for clearing the depth buffer to 1.0. [S]

3.6 Miscellaneous

Avoid "round-trip" calls

Calls such as glGetFloatv, glGetIntegerv, glIsEnabled, glGetError, glGetString require a slow, round trip transaction between the application and renderer. Especially avoid them in your main rendering code.

Note that software implementations of OpenGL may actually perform these operations faster than hardware systems. If you're developing on a low-end system be aware of this fact. [H,L]

Avoid glPushAttrib

If only a few pieces of state need to be saved and restored it's often faster to maintain the information in the client program. glpushAttrib(GL_ALL_ATTRIB_BITS) in particular can be very expensive on hardware systems. This call may be faster in software implementations than in hardware. [H,L]

Check for GL errors during development

During development call glGetError inside your rendering/event loop to catch errors. GL errors raised during rendering can slow down rendering speed. Remove the glGetError call for production code since it's a "round trip" command and can cause delays. [all]

Use glColorMaterial instead of glMaterial

If you need to change a material property on a per vertex basis, glColorMaterial may be faster than glMaterial. [all]

glDrawPixels

- glDrawPixels often performs best with GL_UNSIGNED_BYTE color components [all]
- Disable all unnecessary raster operations before calling glDrawPixels. [all]
- Use the GL_EXT_abgr extension to specify color components in alpha, blue, green, red order on systems which were designed for IRIS GL. [H,L].
- Avoid using viewports which are larger than the window

Software implementations may have to do additional clipping in this situation. [S]

Alpha planes

Don't allocate alpha planes in the color buffer if you don't need them. Specifically, they are not needed for transparency effects. Systems without hardware alpha planes may have to resort to a slow software implementation. [L,S]

Accumulation, stencil, overlay planes

Do not allocate accumulation, stencil or overlay planes if they are not needed. [all]

Be aware of the depth buffer's depth

Your OpenGL may support several different sizes of depth buffers - 16 and 24-bit for example. Shallower depth buffers may be faster than deep buffers both for software and hardware implementations. However, the precision of of a 16-bit depth buffer may not be sufficient for some applications. [L,S]

Transparency may be implemented with stippling instead of blending

If you need simple transparent objects consider using polygon stippling instead of alpha blending. The later is typically faster and may actually look better in some situations. [L,S]

Group state changes together

Try to mimimize the number of GL state changes in your code. When GL state is changed, internal state may have to be recomputed, introducing delays. [all]

Avoid using glPolygonMode

If you need to draw many polygon outlines or vertex points use glBegin with GL_POINTS, GL_LINES, GL_LINE_LOOP or GL_LINE_STRIP instead as it can be much faster. [all]

3.7 Window System Integration

Minimize calls to the make current call

The glXMakeCurrent call, for example, can be expensive on hardware systems because the context switch may involve moving a large amount of data in and out of the hardware.

Visual / pixel format performance

Some X visuals or pixel formats may be faster than others. On PCs for example, 24-bit color buffers may be slower to read/write than 12 or 8-bit buffers. There is often a tradeoff between performance and quality of frame buffer configurations. 12-bit color may not look as nice as 24-bit color. A 16-bit depth buffer won't have the precision of a 24-bit depth buffer.

The GLX_EXT_visual_rating extension can help you select visuals based on performance or quality. GLX 1.2's *visual caveat* attribute can tell you if a visual has a performance penalty associated with it.

It may be worthwhile to experiment with different visuals to determine if there's any advantage of one over another.

Avoid mixing OpenGL rendering with native rendering

OpenGL allows both itself and the native window system to render into the same window. For this to be done correctly synchronization is needed. The GLX glxWaitx and glxWaitGL functions serve this purpose.

Synchronization hurts performance. Therefore, if you need to render with both OpenGL and native window system calls try to group the rendering calls to minimize synchronization.

For example, if you're drawing a 3-D scene with OpenGL and displaying text with X, draw all the 3-D elements first, call <code>glxWaitGL</code> to synchronize, then call all the X drawing functions.

Don't redraw more than necessary

Be sure that you're not redrawing your scene unnecissarily. For example, expose/repaint events may come in batches describing separate regions of the window which must be redrawn. Since one usually redraws the whole window image with OpenGL you only need to respond to one expose/repaint event. In the case of X, look at the count field of the XExposeEvent structure. Only redraw when it is zero.

Also, when responding to mouse motion events you should skip extra motion events in the input queue. Otherwise, if you try to process every motion event and redraw your scene there will be a noticable delay between mouse input and screen updates.

It can be a good idea to put a print statement in your redraw and event loop function so you know exactly what messages are causing your scene to be redrawn, and when.

SwapBuffer calls and graphics pipe blocking

On systems with 3-D graphics hardware the SwapBuffers call is synchronized to the monitor's vertical retrace. Input to the OpenGL command queue may be blocked until the buffer swap has completed. Therefore, don't put more OpenGL calls immediately after SwapBuffers. Instead, put application computation instructions which can overlap with the buffer swap delay.

3.8 Mesa-specific

Mesa is a free library which implements most of the OpenGL API in a compatible manner. Since it is a software library, performance depends a great deal on the host computer. There are several Mesa-specific features to be aware of which can effect performance.

Double buffering

The X driver supports two back color buffer implementations: Pixmaps and XImages. The MESA_BACK_BUFFER environment variable controls which is used. Which of the two that's faster depends on the nature of your rendering. Experiment.

X Visuals

As described above, some X visuals can be rendered into more quickly than others. The MESA_RGB_VISUAL environment variable can be used to determine the quickest visual by experimentation.

Depth buffers

Mesa may use a 16 or 32-bit depth buffer as specified in the src/config.h configuration file. 16-bit depth buffers are faster but may not offer the precision needed for all applications.

Flat-shaded primitives

If one is drawing a number of flat-shaded primitives all of the same color the glColor command should be put before the glBegin call.

Don't do this:

```
glBegin(...);
glColor(...);
glVertex(...);
...
glEnd();
```

Do this:

```
glColor(...);
glBegin(...);
glVertex(...);
...
glEnd();
```

glColor*() commands

The glColor[34]ub[v] are the fastest versions of the glColor command.

Avoid double precision valued functions

Mesa does all internal floating point computations in single precision floating point. API functions which take double precision floating point values must convert them to single precision. This can be expensive in the case of glVertex, glNormal, etc.

4. Evaluation and Tuning

To maximize the performance of an OpenGL applications one must be able to evaluate an application to learn what is limiting its speed. Because of the hardware involved it's not sufficient to use ordinary profiling tools. Several different aspects of the graphics system must be evaluated.

Performance evaluation is a large subject and only the basics are covered here. For more information see "OpenGL on Silicon Graphics Systems".

4.1 Pipeline tuning

The graphics system can be divided into three subsystems for the purpose of performance evaluation:

- CPU subsystem application code which drives the graphics subsystem
- Geometry subsystem transformation of vertices, lighting, and clipping
- Rasterization subsystem drawing filled polygons, line segments and per-pixel processing

At any given time, one of these stages will be the bottleneck. The bottleneck must be reduced to improve performance. The strategy is to isolate each subsystem in turn and evaluate changes in performance. For example, by decreasing the workload of the CPU subsystem one can determine if the CPU or graphics system is limiting performance.

4.1.1 CPU subsystem

To isosulate the CPU subsystem one must reduce the graphics workload while presevering the application's execution characteristics. A simple way to do this is to replace glVertex() and glNormal calls with glColor calls. If performance does not improve then the CPU stage is the bottleneck.

4.1.2 Geometry subsystem

To isoslate the geometry subsystem one wants to reduce the number of primitives processed, or reduce the transformation work per primitive while producing the same number of pixels during rasterization. This can be done by replacing many small polygons with fewer large ones or by simply disabling lighting or clipping. If performance increases then your application is bound by geometry/transformation speed.

4.1.3 Rasterization subsystem

A simple way to reduce the rasterization workload is to make your window smaller. Other ways to reduce rasterization work is to disable per-pixel processing such as texturing, blending, or depth testing. If performance increases, your program is *fill limited*.

After bottlenecks have been identified the techniques outlined in section 3 can be applied. The process of identifying and reducing bottlenecks should be repeated until no further improvements can be made or your minimum performance threshold has been met.

4.2 Double buffering

For smooth animation one must maintain a high, constant frame rate. Double buffering has an important effect on this. Suppose your application needs to render at 60Hz but is only getting 30Hz. It's a mistake to think that you must reduce rendering time by 50% to achive 60Hz. The reason is the swap-buffers operation is synchronized to occur during the display's vertical retrace period (at 60Hz for example). It may be that your application is taking only a tiny bit too long to meet the 1/60 second rendering time limit for 60Hz.

Measure the performance of rendering in single buffer mode to determine how far you really are from your target frame rate.

4.3 Test on several implementations

The performance of OpenGL implementations varies a lot. One should measure performance and test OpenGL applications on several different systems to be sure there are no unexpected problems.

Last edited on May 16, 1997 by Brian Paul.



Mark J. Kilgard <u>mjk@nvidia.com</u> NVIDIA Corporation

Copyright 1998, 1999 by Mark J. Kilgard. Commercial publication in written, electronic, or other forms without expressed written permission is prohibited. Electronic redistribution for educational or private use is permitted.

Every software engineer who has programmed long enough has a war story about some insidious bug that induced head scratching, late night debugging, and probably even schedule delays. More often than we programmers care to admit, the bug turns out to be self-inflicted. The difference between an experienced programmer and a novice is knowing the good practices to use and the bad practices to avoid so those self-inflicted bugs are kept to a minimum.

A programming interface pitfall is a self-inflicted bug that is the result of a misunderstanding about how a particular programming interface behaves. The pitfall may be the fault of the programming interface itself or its documentation, but it is often simply a failure on the programmer's part to fully appreciate the interface's specified behavior. Often the same set of basic pitfalls plagues novice programmers because they simply have not yet learned the intricacies of a new programming interface.

You can learn about the programming interface pitfalls in two ways: The hard way and the easy way. The hard way is to experience them one by one, late at night, and with a deadline hanging over your head. As a wise main once explained, "Experience is a good teacher, but her fees are very high." The easy way is to benefit from the experience of others.

This is your opportunity to learn how to avoid 16 software pitfalls common to beginning and intermediate OpenGL programmers. This is your chance to spend a bit of time reading now to avoid much grief and frustration down the line. I will be honest; many of these pitfalls I learned the hard way instead of the easy way. If you program OpenGL seriously, I am confident that the advice below will make you a better OpenGL programmer.

If you are a beginning OpenGL programmer, some of the discussion below might be about topics that you have not yet encountered. This is not the place for a complete introduction to some of the more complex OpenGL topics covered such as mipmapped texture mapping or OpenGL's pixel transfer modes. Feel free to simply skim over sections that may be too advanced. As you develop as an OpenGL programmer, the advice will become more worthwhile.

1. Improperly Scaling Normals for Lighting

Enabling lighting in OpenGL is a way to make your surfaces appear more realistic. Proper use of OpenGL's lighting model provides subtle clues to the viewer about the curvature and orientation of surfaces in your scene.

When you render geometry with lighting enabled, you supply normal vectors that indicate the orientation of the surface at each vertex. Surface normals are used when calculating diffuse and specular lighting effects. For example, here is a single rectangular patch that includes surface normals:

```
glBegin(GL_QUADS);
glNormal3f(0.181636,-0.25,0.951057);
glVertex3f(0.549,-0.756,0.261);
glNormal3f(0.095492,-0.29389,0.95106);
glVertex3f(0.288,-0.889,0.261);
glNormal3f(0.18164,-0.55902,0.80902);
glVertex3f(0.312,-0.962,0.222);
glNormal3f(0.34549,-0.47553,0.80902);
glVertex3f(0.594,-0.818,0.222);
glEnd();
```

The x, y, and z parameters for each glNormal3f call specify a direction vector. If you do the math, you will find that the length of each normal vector above is essentially 1.0. Using the first glNormal3f call as an example, observe that:

 $sqrt(0.181636^2 + -0.25^2 + 0.951057^2) \approx 1.0$

For OpenGL's lighting equations to operate properly, the assumption OpenGL makes by default is that the normals passed to it are vectors of length 1.0.

However, consider what happens if before executing the above OpenGL primitive, glscalef is used to shrink or enlarge subsequent OpenGL geometric primitives. For example:

```
glMatrixMode(GL_MODELVIEW);
glScalef(3.0, 3.0, 3.0);
```

The above call causes subsequent vertices to be enlarged by a factor of three in each of the *x*, *y*, and *z* directions by scaling OpenGL's modelview matrix. gllscalef can be useful for enlarging or shrinking geometric objects, but you must be careful because OpenGL transforms normals using a version of the modelview matrix called the *inverse transpose modelview* matrix. Any enlarging or shrinking of vertices during the modelview transformation *also* changes the length of normals.

Here is the pitfall: Any modelview scaling that occurs is likely to mess up OpenGL's lighting equations. Remember, the lighting equations assume that normals have a length of 1.0. The symptom of incorrectly scaled normals is that the lit surfaces appear too dim or too bright depending on whether the normals enlarged or shrunk.

The simplest way to avoid this pitfall is by calling:

glEnable(GL_NORMALIZE);

This mode is not enabled by default because it involves several additional calculations. Enabling the mode forces OpenGL to normalize transformed normals to be of unit length before using the normals in OpenGL's lighting equations. While this corrects potential lighting problems introduced by scaling, it also slows OpenGL's vertex processing speed since normalization requires extra operations, including several multiplies and an expensive reciprocal square root operation. While you may argue whether this mode should be enabled by default or not, OpenGL's designers thought it better to make the default case be the fast one. Once you are aware of the need for this mode, it is easy to enable when you know you need it.

There are two other ways to avoid problems from scaled normals that may let you avoid the performance penalty of enabling $GL_NORMALIZE$. One is simply to not use glscalef to scale vertices. If you need to scale vertices, try scaling the vertices before sending them to OpenGL. Referring to the above example, if the application simply multiplied each glVertex3f by 3, you could eliminate the need for the above glscalef without having the enable the $GL_NORMALIZE$ mode.

Note that while glscalef is problematic, you can safely use glTranslatef and glRotatef because these routines change the modelview matrix transformation without introducing any scaling effects. Also, be aware that glMatrixMultf can also be a source of normal scaling problems if the matrix you multiply by introduces scaling effects.

The other option is to adjust the normal vectors passed to OpenGL so that after the inverse transpose modelview transformation, the resulting normal will become a unit vector. For example, if the earlier <code>glScalef</code> call tripled the vertex coordinates, we could correct for this corresponding thirding effect on the transformed normals by pre-multiplying each normal component by 3.

OpenGL 1.2 adds a new glEnable mode called GL_RESCALE_NORMAL that is potentially more efficient than the GL_NORMALIZE mode. Instead of performing a true normalization of the transformed normal vector, the transformed normal vector is scaled based on a scale factor computed from the inverse modelview matrix as diagonal terms. GL_RESCALE_NORMAL can be used when the modelview matrix has a uniform scaling factor.

2. Poor Tessellation Hurts Lighting

OpenGL's lighting calculations are done *per-vertex*. This means that the shading calculations due to light sources interacting with the surface material of a 3D object are only calculated at the object's vertices. Typically, OpenGL just interpolates or *smooth shades* between vertex colors. OpenGL's per-vertex lighting works pretty well except when a lighting effect such as a specular highlight or a spotlight is lost or blurred because the effect is not sufficiently sampled by an object's vertices. Such under -sampling of lighting effects occurs when objects are coarsely modeled to use a minimal number of vertices.

Figure 1 shows an example of this problem. The top left and top right cubes each have an identically configured OpenGL spotlight light source shining directly on each cube. The left cube has a nicely defined spotlight pattern; the right cube lacks any clearly defined spotlight pattern. The key difference between the two models is the number of vertices used to model each cube. The left cube models each surface with over 120 distinct vertices; the right cube has only 4 vertices.

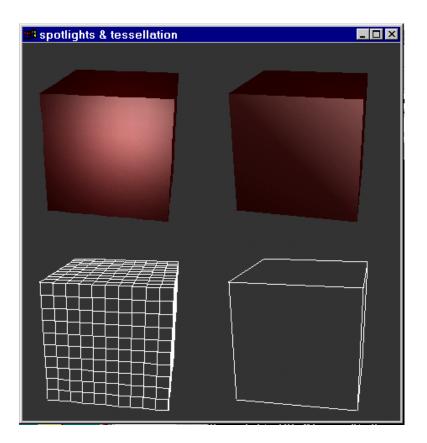


Figure 1: Two cubes rendered with identical OpenGL spotlight enabled. (The lines should all be connected but are not due to resampling in the image above.)

At the extreme, if you tessellate the cube to the point that each polygon making up the cube is no larger than a pixel, the lighting effect will essentially become per-pixel. The problem is that the rendering will probably no longer be interactive. One good thing about per-vertex lighting is that you decide how to trade off rendering speed for lighting fidelity.

Smooth shading between lit vertices helps when the color changes are gradual and fairly linear. The problem is that effects such as spotlights, specular highlights, and non-linear light source attenuation are often not gradual. OpenGL's lighting model only does a good job capturing these effects if the objects involved are reasonably tessellated.

Novice OpenGL programmers are often tempted to enable OpenGL's spotlight functionality and shine a spotlight on a wall modeled as a single huge polygon. Unfortunately, no sharp spotlight pattern will appear as the novice intended; you probably will not see any spotlight affect at all. The problem is that the spotlight's cutoff means that the extreme corners of the wall where the vertices are specified get *no* contribution from the spotlight and since those are the only vertices the wall has, there will be no spotlight pattern on the wall.

If you use spotlights, make sure that you have sufficiently tessellated the lit objects in your scene with enough vertices to capture the spotlight effect. There is a speed/quality tradeoff here: More vertices mean better lighting effects, but also increases the amount of vertex transformation required to render the scene.

Specular highlights (such as the bright spot you often see on a pool ball) also require sufficiently tessellated objects to capture the specular highlight well.

Keep in mind that if you use more linear lighting effects such as ambient and diffuse lighting effects where there are typically not *sharp* lighting changes, you can get good lighting effects with even fairly coarse tessellation.

If you do want *both* high quality and high -speed lighting effects, one option is to try using multi-pass texturing techniques to texture specular highlights and spotlight patterns onto objects in your scene. Texturing is a per-fragment operation so you can correctly capture per -fragment lighting effects. This can be involved, but such techniques can deliver fast, high-quality lighting effects when used effectively.

3. Remember Your Matrix Mode

OpenGL has a number of 4 by 4 matrices that control the transformation of vertices, normals, and texture coordinates. The core OpenGL standard specifies the modelview matrix, the projection matrix, and the texture matrix.

Most OpenGL programmers quickly become familiar with the modelview and projection matrices. The modelview matrix controls the viewing and modeling transformations for your scene. The projection matrix defines the view frustum and controls the how the 3D scene is projected into a 2D image. The texture matrix may be unfamiliar to some; it allows you to transform texture coordinates to accomplish effects such as projected textures or sliding a texture image across a geometric surface.

A single set of matrix manipulation commands controls all types of OpenGL matrices: glscalef, glTranslatef, glRotatef, glLoadIdentity, glMultMatrixf, and several other commands. For efficient saving and restoring of matrix state, OpenGL provides the glPushMatrix and glPopMatrix commands; each matrix type has its own a stack of matrices.

None of the matrix manipulation commands have an explicit parameter to control which matrix they affect. Instead, OpenGL maintains a current matrix mode that determines which matrix type the previously mentioned matrix manipulation commands actually affects. To change the matrix mode, use the glMatrixMode command. For example:

```
glMatrixMode(GL_PROJECTION);
/* Now update the projection matrix. */
glLoadIdentity();
glFrustum(-1, 1, -1, 1, 0.0, 40.0);
glMatrixMode(GL_MODELVIEW);
/* Now update the modelview matrix. */
glPushMatrix();
glRotatef(45.0, 1.0, 1.0, 1.0);
render();
glPopMatrix();
```

A common pitfall is forgetting the current setting of the matrix mode and performing operations on the wrong matrix stack. If later code assumes the matrix mode is set to a particular state, you both fail to update the matrix you intended and screw up whatever the actual current matrix is.

If this can trip up the unwary programmer, why would OpenGL have a matrix mode? Would it not make sense for each matrix manipulation routine to also pass in the matrix that it should manipulate? The answer is simple: lower overhead. OpenGL's design optimizes for the common case. In real programs, matrix manipulations occur more often than matrix mode changes. The common case is a sequence of matrix operations all updating the same matrix type. Therefore, typical OpenGL usage is optimized by controlling which matrix is manipulated based on the current matrix mode. When you call glMatrixMode, OpenGL configures the matrix manipulation commands to efficiently update the current matrix type. This saves time compared to deciding which matrix to update every time a matrix manipulation is performed.

In practice, because a given matrix type does tend to be updated repeatedly before switching to a different matrix, the lower overhead for matrix manipulation more than makes up for the programmer's burden of ensuring the matrix mode is properly set before matrix manipulation.

A simple program-wide policy for OpenGL matrix manipulation helps avoid pitfalls when manipulating matrices. Such a policy would require any code manipulating a matrix to first call <code>glMatrixMode</code> to always update the intended matrix. However in most programs, the modelview matrix is manipulated quite frequently during rendering and the other matrices change considerably less frequently overall. If this is the case, a better policy is that routines can assume the matrix mode is set to update the modelview matrix. Routines that need to update a different matrix are responsible to switch back to the modelview matrix after manipulating one of the other matrices.

Here is an example of how OpenGL's matrix mode can get you into trouble. Consider a program written to keep a constant aspect ratio for an OpenGL-rendered scene in a window. Maintaining the aspect ratio requires updating the projection matrix whenever the window is resized. OpenGL programs typically also adjust the OpenGL viewport in response to a window resize so the code to handle a window resize notification might look like this:

```
void
doResize(int newWidth, int newHieght)
{
  GLfloat aspectRatio = (GLfloat)newWidth / (GLfloat)newHeight;
  glViewport(0, 0, newWidth, newHeight);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, aspectRatio, 0.1, 40.0);
  /* WARNING: matrix mode left as projection! */
}
```

If this code fragment is from a typical OpenGL program, doResize is one of the few times or even only time the projection

matrix gets changed after initialization. This means that it makes sense to add to a final glMatrixMode (GL_MODELVIEW) call to doResize to switch back to the modelview matrix. This allows the window's redraw code safely assume the current matrix mode is set to update the modelview matrix and eliminate a call to glMatrixMode. Since window redraws often repeatedly update the modelview matrix, and redraws occur considerably more frequently than window resizes, this is generally a good approach.

A tempting approach might be to call glGetIntegerv to retrieve the current matrix mode state and then only change the matrix mode when it was not what you need it to be. After performing its matrix manipulations, you could even restore the original matrix mode state.

This is however almost certainly a bad approach. OpenGL is designed for fast rendering and setting state; retrieving OpenGL state is often considerably slower than simply setting the state the way you require. As a rule, glGetIntegerv and related state retrieval routines should only be used for debugging or retrieving OpenGL implementation limits. They should *never* be used in performance critical code. On faster OpenGL implementations where much of OpenGL's state is maintained within the graphics hardware, the relative cost of state retrieval commands is considerably higher than in largely software-based OpenGL implementations. This is because state retrieval calls must stall the graphics hardware to return the requested state. When users run OpenGL programs on high-performance expensive graphics hardware and do not see the performance gains they expect, in many cases the reason is invocations of state retrieval commands that end up stalling the hardware to retrieve OpenGL state.

In cases where you do need to make sure that you restore the previous matrix mode after changing it, try using glPushAttrib with the GL_TRANSFORM_BIT bit set and then use glPopAttrib to restore the matrix mode as needed. Pushing and popping attributes on the attribute stack can be more efficient than reading back the state and later restoring it. This is because manipulating the attribute stack can completely avoid stalling the hardware if the attribute stack exists within the hardware. Still the attribute stack is not particularly efficient since all the OpenGL transform state (including clipping planes and the normalize flag) must also be pushed and popped.

The advice in this section is focused on the matrix mode state, but pitfalls that relate to state changing and restoring are common in OpenGL. OpenGL's explicit state model is extremely well suited to the stateful nature of graphics hardware, but can be an unwelcome burden for programmers not used to managing graphics state. With a little experience though, managing OpenGL state becomes second nature and helps ensure good hardware utilization.

The chief advantage of OpenGL's stateful approach is that well-written OpenGL rendering code can minimize state changes so that OpenGL can maximize rendering performance. A graphics- interface that tries to hide the inherently stateful nature of well-designed graphics hardware ends up either forcing redundant state changes or adds extra overhead by trying to eliminate such redundant state changes. Both approaches give up performance for convenience. A smarter approach is relying on the application or a high-level graphics library to manage graphics state. Such a high-level approach is typically more efficient in its utilization of fast graphics hardware when compared to attempts to manage graphics state in a low-level library without high-level knowledge of how the operations are being used.

If you want more convenient state management, consider using a high-level graphics library such as Open Inventor or IRIS Performer that provide both a convenient programming model and efficient high-level management of OpenGL state changes.

4. Overflowing the Projection Matrix Stack

OpenGL's glPushMatrix and glPopMatrix commands make it very easy to perform a set of cumulative matrix operations, do rendering, and then restore the matrix state to that before the matrix operations and rendering. This is very handy when doing hierarchical modeling during rendering operations.

For efficiency reasons and to permit the matrix stacks to exist within dedicated graphics hardware, the size of OpenGL's various matrix stacks are limited. OpenGL mandates that all implementations *must* provide at least a 32-entry modelview matrix stack, a 2-entry projection matrix stack, and a 2-entry texture matrix stack. Implementations are free to provide larger stacks, and glGetIntergerv provides a means to query an implementation's actual maximum depth.

Calling glPushMatrix when the current matrix mode stack is already at its maximum depth generates a GL_STACK_UNDERFLOW error and the responsible glPushMatrix is ignored. OpenGL applications guaranteed to run correctly on all OpenGL implementations should respect the minimum stack limits cited above (or better yet, query the implementation's true stack limit and respect that).

This can become a pitfall when software-based OpenGL implementations implement stack depth limits that exceed the minimum limits. Because these stacks are maintained in general purpose memory and not within dedicated graphics hardware, there is no substantial expense to permitting larger or even unlimited matrix stacks as there is when the matrix stacks are implemented in dedicated hardware. If you write your OpenGL program and test it against such implementations with large or unlimited stack sizes, you may not notice that you exceeded a matrix stack limit that would exist on an OpenGL implementation that only implemented OpenGL's mandated minimum stack limits.

The 32 required modelview stack entries will not be exceeded by most applications (it can still be done so be careful). However, programmers should be on guard not to exceed the projection and texture matrix stack limits since these stacks may have as few as 2 entries. In general, situations where you actually need a projection or texture matrix that exceed two entries are quite rare and generally avoidable.

Consider this example where an application uses two projection matrix stack entries for updating a window:

```
void
renderWindow(void)
{
 render3Dview();
 glPushMatrix();
   glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
      glLoadIdentity();
      gluOrtho2D(0, 1, 0, 1);
      render2Doverlay();
    glPopMatrix();
  glPopMatrix();
  glMatrixMode(GL_MODELVIEW);
}
```

The window renders a 3D scene with a 3D perspective projection matrix (initialization not shown), then switches to a simple 2D orthographic projection matrix to draw a 2D overlay.

Be careful because if the render2Doverlay tries to push the projection matrix again, the projection matrix stack will overflow on some machines. While using a matrix push, cumulative matrix operations, and a matrix pop is a natural means to accomplish hierarchical modeling, the projection and texture matrices rarely require this capability. In general, changes to the projection matrix are to switch to an entirely different view (not to make a cumulative matrix change to later be undone). A simple matrix switch (reload) does not need a push and pop stack operation.

If you find yourself attempting to push the projection or texture matrices beyond two entries, consider if there is a simpler way to accomplish your manipulations that will not overflow these stacks. If not, you are introducing a latent interoperability problem when you program is run on high-performance hardware-intensive OpenGL implementations that implement limited projection and texture matrix stacks.

5. Not Setting All Mipmap Levels

When you desire high-quality texture mapping, you will typically specify a mipmapped texture filter. Mipmapping lets you specify multiple levels of detail for a texture image. Each level of detail is half the size of the previous level of detail in each dimension. So if your initial texture image is an image of size 32x32, the lower levels of detail will be of size 16x16, 8x8, 4x4, 2x2, and 1x1. Typically, you use the gluBuild2DMipmaps routine to automatically construct the lower levels of details from you original image. This routine re-samples the original image at each level of detail so that the image is available at each of the various smaller sizes.

Mipmap texture filtering means that instead of applying texels from a single high-resolution texture image, OpenGL automatically selects from the best pre-filtered level of detail. Mipmapping avoids distracting visual artifacts that occur when a distant textured object under-samples its associated texture image. With a mipmapped minimization filter enabled, instead of under-sampling a single high resolution texture image, OpenGL will automatically select the most appropriate levels of detail.

One pitfall to be aware of is that if you do not specify every necessary level of detail, OpenGL will silently act as if texturing is not enabled. The OpenGL specification is very clear about this: "If texturing is enabled (and TEXTURE_MIN_FILTER is one that requires a mipmap) at the time a primitive is rasterized and if the set of arrays 0 through *n* is incomplete, based on the dimensions of array 0, then it is as if texture mapping were disabled."

The pitfall typically catches you when you switch from using a non-mipmapped texture filter (like GL_LINEAR) to a mipmapped filter, but you forget to build complete mipmap levels. For example, say you enabled non-mipmapped texture mapping like this:

```
glEnable(GL_TEXTURE_2D);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 3, width, height, GL_RGB, GL_UNSIGNED_BYTE,
imageData);
```

At this point, you could render non-mipmapped textured primitives. Where you could get tripped up is if you naively simply enabled a mipmapped minification filter. For example:

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);

The problem is that you have changed the minification filter, but not supplied a complete set of mipmap levels. Not only do you not get the filtering mode you requested, but also subsequent rendering happens as if texture mapping were not even enabled.

The simple way to avoid this pitfall is to use gluBuild2DMipmaps (or gluBuild1DMipmaps for 1D texture mapping) whenever you are planning to use a mipmapped minification filter. So this works:

glEnable(GL_TEXTURE_2D); glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR); gluBuild2DMipmaps(GL_TEXTURE_2D, depth, width, height, GL_RGB, GL_UNSIGNED_BYTE, imageData);

The above code uses a mipmap filter and uses gluBuild2DMipmaps to make sure all the levels are populated correctly. Subsequent rendering is not just textured, but properly uses mipmapped filtering.

Also, understand that OpenGL considers the mipmap levels incomplete not simply because you have not specified all the mipmap levels, but also if the various mipmap levels are inconsistent. This means that you must consistently specify border pixels and each successive level must be half the size of the previous level in each dimension.

6. Reading Back Luminance Pixels

You can use OpenGL's glReadPixels command to read back rectangular regions of a window into your program's memory space. While reading back a color buffer as RGB or RGBA values is straightforward, OpenGL also lets you read back luminance values, but it can a bit tricky to get what you probably expect. Retrieving luminance values is useful if you want to generate a grayscale image.

When you read back luminance values, the conversion to luminance is done as a simple addition of the distinct red, green, and blue components with result clamped between 0.0 and 1.0. There is a subtle catch to this. Say the pixel you are reading back is 0.5 red, 0.5 green, and 0.5 blue. You would expect the result to then be a medium gray value. However, just adding these components would give 1.5 that would be clamped to 1.0. Instead of being a luminance value of 0.5, as you would expect, you get pure white.

A naive reading of luminance values results in a substantially brighter image than you would expect with a high likelihood of many pixels being saturated white.

The right solution would be to scale each red, green, and blue component appropriately. Fortunately, OpenGL's pixel transfer operations allow you to accomplish this with a great deal of flexibility. OpenGL lets you scale and bias each component separately when you send pixel data through OpenGL.

For example, if you wanted each color component to be evenly averaged during pixel read back, you would change OpenGL's default pixel transfer state like this:

glPixelTransferf(GL_RED_SCALE,0.3333); glPixelTransferf(GL_GREEN_SCALE,0.3334); glPixelTransferf(GL_BLUE_SCALE,0.3333);

With OpenGL's state set this way, glReadPixels will have cut each color component by a third before adding the components during luminance conversion. In the previous example of reading back a pixel composed of 0.5 red, 0.5 green, and 0.5 blue, the resulting luminance value is 0.5.

However, as you may be aware, your eye does not equally perceive the contribution of the red, green, and blue color components. A standard linear weighting for combining red, green, and blue into luminance was defined by the National Television Standard Committee (NTSC) when the US color television format was standardized. These weightings are based on the human eye's sensitivity to different wavelengths of visible light and are based on extensive research. To set up OpenGL to convert RGB to luminance according to the NTSC standard, you would change OpenGL's default pixel transfer state like this:

glPixelTransferf(GL_RED_SCALE, 0.299);

glPixelTransferf(GL_GREEN_SCALE, 0.587);
glPixelTransferf(GL_BLUE_SCALE, 0.114);

If you are reading back a luminance version of an RGB image that is intended for human viewing, you probably will want to use the NTSC scale factors.

Something to appreciate in all this is how OpenGL itself does not mandate a particular scale factor or bias for combining color components into a luminance value; instead, OpenGL's flexible pixel path capabilities give the application control. For example, you could easily read back a luminance image where you had suppressed any contribution from the green color component if that was valuable to you by setting the green pixel transfer scale to be 0.0 and re-weighting red and blue appropriately.

You could also use the biasing capability of OpenGL's pixel transfer path to enhance the contribution of red in your image by adding a bias like this:

glPixelTransferf(GL_RED_BIAS, 0.1);

That will add 0.1 to each red component as it is read back.Please note that the default scale factor is 1.0 and the default bias is 0.0. Also be aware that these same modes are not simply used for the luminance read back case, but *all* pixel or texture copying, reading, or writing. If you program changes the scales and biases for reading luminance values, it will probably want to restore the default pixel transfer modes when downloading textures.

7. Watch Your Pixel Store Alignment

OpenGL's pixel store state controls how a pixel rectangle or texture is read from or written to your application's memory. Consider what happens when you call glDrawPixels. You pass a pointer to the pixel rectangle to OpenGL. But how exactly do pixels in your application's linear address space get turned into an image?

The answer sounds like it should be straightforward. Since glDrawPixels takes a width and height in pixels and a (that implies some number of bytes per pixel), you could just assume the pixels were all packed in a tight array based on the parameters passed to glDrawPixels. Each row of pixels would immediately follow the previous row.

In practice though, applications often need to extract a sub-rectangle of pixels from a larger packed pixel rectangle. Or for performance reasons, each row of pixels is setup to begin on some regular byte alignment. Or the pixel data was read from a file generated on a machine with a different byte order (Intel and DEC processors are little -endian; Sun, SGI, and Motorola processors are big-endian).

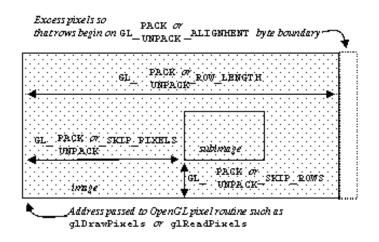


Figure 2: Relationship of the image layout pixel store modes.

So OpenGL's pixel store state determines how bytes in your application's address space get unpacked from or packed to OpenGL images. Figure 2 shows how the pixel state determines the image layout. In addition to the image layout, other pixel store state determines the byte order and bit ordering for pixel data.

One likely source of surprise for OpenGL programmers is the default state of the GL_PACK_ALIGNMENT and GL_UNPACK_ALIGNMENT values. Instead of being 1, meaning that pixels are packed into rows with no extra bytes between rows, the actual default for these modes is 4.

Say that your application needs to read back an 11 by 8 pixel area of the screen as RGB pixels (3

bytes per pixel, one byte per color component). The following glReadPixels call would read the pixels:

```
glReadPixels(x, y, 11, 8, GL_RGB, GL_UNSIGNED_BYTE, pixels);
```

How large should the pixels array need to be to store the image? Assume that the GL_UNPACK_ALIGNMENT state is still 4 (the initial value). Naively, your application might call:

pixels = (GLubyte*) malloc(3 * 11 * 8); /* Wrong! */

Unfortunately, the above code is wrong since it does not account for OpenGL's default 4-byte row alignment. Each row of pixels will be 33 bytes wide, but then each row is padded to be 4 byte aligned. The effective row width in bytes is then 36. The above malloc call will not allocate enough space; the result is that glReadPixels will write several pixels beyond the allocated range and corrupt memory.

With a 4 byte row alignment, the actual space required is not *simply BytesPerPixel* \hat{Width} *Height*, but instead ((*BytesPerPixel* $\hat{Width} + 3) >> 2) << 2)$ \hat{Height} . Despite the fact that OpenGL's initial pack and unpack alignment state is 4, most programs should not use a 4 byte row alignment and instead request that OpenGL tightly pack and unpack pixel rows. To avoid the complications of excess bytes at the end of pixel rows for alignment, change OpenGL's row alignment state to be "tight" like this:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glPixelStorei(GL_PACK_ALIGNMENT, 1);
```

Be extra cautious when your program is written assuming a 1 byte row alignment because bugs caused by OpenGL's initial 4 byte row alignment can easily go unnoticed. For example, if such a program is tested only with images and textures of width divisible by 4, no memory corruption problem is noticed since the test images and textures result in a tight row packing. And because lots of textures and images, by luck or design, have a width divisible by 4, such a bug can easily slip by your testing. However, the memory corruption bug is bound to surface as soon as a customer tries to load a 37 pixel width image.

Unless you *really* want a row alignment of 4, be sure you change this state when using pixel rectangles, 2D and 1D textures, bitmaps, and stipple patterns. And remember that there is a distinct pack and unpack row alignment.

8. Know Your Pixel Store State

Keep in mind that your pixel store state gets used for textures, pixel rectangles, stipple patterns, and bitmaps. Depending on what sort of 2D image data you are passing to (or reading back from) OpenGL, you may need to load the pixel store unpack (or pack) state.

Not properly configuring the pixel store state (as described in the previous section) is one common pitfall. Yet another pitfall is changing the pixel store modes to those needed by a particular OpenGL commands and later issuing some other OpenGL commands requiring the original pixel store mode settings. To be on the safe side, it is usually a good idea to save and restore the previous pixel store modes when you need to change them.

Here is an example of such a save and restore. The following code saves the pixel store unpack modes:

```
GLint swapbytes, lsbfirst, rowlength, skiprows, skippixels, alignment;
/* Save current pixel store state. */
glGetIntegerv(GL_UNPACK_SWAP_BYTES, &swapbytes);
glGetIntegerv(GL_UNPACK_LSB_FIRST, &lsbfirst);
glGetIntegerv(GL_UNPACK_ROW_LENGTH, &rowlength);
glGetIntegerv(GL_UNPACK_SKIP_ROWS, &skiprows);
glGetIntegerv(GL_UNPACK_SKIP_PIXELS, &skippixels);
glGetIntegerv(GL_UNPACK_ALIGNMENT, &alignment);
/* Set desired pixel store state. */
glPixelStorei(GL_UNPACK_SWAP_BYTES, GL_FALSE);
glPixelStorei(GL_UNPACK_LSB_FIRST, GL_FALSE);
glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
glPixelStorei(GL_UNPACK_SKIP_ROWS, 0);
glPixelStorei(GL_UNPACK_SKIP_PIXELS, 0);
```

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

Then, this code restores the pixel store unpack modes:

```
/* Restore current pixel store state. */
glPixelStorei(GL_UNPACK_SWAP_BYTES, swapbytes);
glPixelStorei(GL_UNPACK_LSB_FIRST, lsbfirst);
glPixelStorei(GL_UNPACK_ROW_LENGTH, rowlength);
glPixelStorei(GL_UNPACK_SKIP_ROWS, skiprows);
glPixelStorei(GL_UNPACK_SKIP_PIXELS, skippixels);
glPixelStorei(GL_UNPACK_ALIGNMENT, alignment);
```

Similar code could be written to save and restore OpenGL's pixel store pack modes (change UNPACK to PACK in the code above).

With OpenGL 1.1, the coding effort to save and restore these modes is simpler. To save, the pixel store state, you can call:

glPushClientAttrib(GL_CLIENT_PIXEL_STORE_BIT);

Then, this code restores the pixel store unpack modes:

glPopClientAttrib(GL_CLIENT_PIXEL_STORE_BIT);

The above routines (introduced in OpenGL 1.1) save and restore the pixel store state by pushing and popping the state using a stack maintained within the OpenGL library.

Observant readers may wonder why glPushClientAttrib is used instead of the shorter glPushAttrib routine. The answer involves the difference between OpenGL client-side and server-side state. It is worth clearly understanding the practical considerations that surround the distinction between OpenGL's server-side and client-side state.

There is not actually the option to use glPushAttrib to push the pixel store state because glPushAttrib and glPopAttrib only affects the server-state attribute stack and the pixel pack and unpack pixel store state is client-side OpenGL state.

Think of your OpenGL application as a client of the OpenGL rendering service provided by the host computer's OpenGL implementation.

The pixel store modes are *client-side state*. However, most of OpenGL's state is server-side. The term server-side state refers to the fact that the state actually resides within the OpenGL implementation itself, possibly within the graphics hardware itself. Server-side OpenGL state is concerned with how OpenGL commands are rendered, but client-side OpenGL state is concerned with how image or vertex data is extracted from the application address space.

Server-side OpenGL state is often expensive to retrieve because the state may reside only within the graphics hardware. To return such hardware-resident state (for example with glGetIntegerv) requires all preceding graphics commands to be issued before the state is retrievable. While OpenGL makes it possible to read back nearly all OpenGL server-side state, well-written programs should always avoid reading back OpenGL server-side state in performance sensitive situations.

Client-side state however is not state that will ever reside only within the rendering hardware. This means that using glGetIntegerv to read back pixel store state is relatively inexpensive because the state is client-side. This is why the above code that explicitly reads back each pixel store unpack mode can be recommended. Similar OpenGL code that tried to save and restore server-side state could severely undermine OpenGL rendering performance.

Consider that whether it is better to use glGetIntegerv and glPixelStorei to explicitly save and restore the modes or whether you use OpenGL 1.1's glPushClientAttrib and glPopClientAttrib will depend on your situation. When pushing and popping the client attribute stack, you do have to be careful not to overflow the stack. An advantage to pushing and popping the client attribute state is that both the pixel store and vertex array client-side state can be pushed or popped with a single call. Still, you may find that only the pack or only the unpack modes need to be saved and restored and sometimes only one or two of the modes. If that is the case, an explicit save and restore may be faster.

9. Careful Updating that Raster Position

OpenGL's raster position determines where pixel rectangles and bitmaps will be rasterized. The glRasterPos2f family

of commands specifies the coordinates for the raster position. The raster position gets transformed just as if it was a vertex. This symmetry makes it easy to position images or text within a scene along side 3D geometry. Just like a vertex, the raster position is logically an (x,y,z,w) coordinate. It also means that when the raster position is specified, OpenGLâs modelview and projection matrix transformations, lighting, clipping, and even texture coordinate generation are all performed on the raster position vertex in exactly the same manner as a vertex coordinate passed to OpenGL via glVertex3f.

While this is all very symmetric, it rarely if ever makes sense to light or generate a texture coordinate for the raster position. It can even be quite confusing when you attempt to render a bitmap based on the current color and find out that because lighting is enabled, the bitmap color gets determined by lighting calculations. Similarly, if you draw a pixel rectangle with texture mapping enabled, your pixel rectangle may end up being modulated with the single texel determined by the current raster texture coordinate.

Still another symmetric, but generally unexpected result of OpenGL's identical treatment of vertices and the raster position is that, just like a vertex, the raster position can be clipped. This means if you specify a raster position outside (even slightly outside) the view frustum, the raster position is clipped and marked "invalid". When the raster position is invalid, OpenGL simply discards the pixel data specified by the glBitmap, glDrawPixels, and glCopyPixls commands.

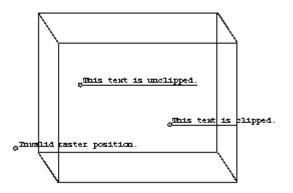


Figure 3: The enclosing box represents the view frustum and viewport. Each line of text is preceded by a dot indicating where the raster position is set before rendering the line of text. The dotted underlining shows the pixels that will actually be rasterized from each line of text. Notice that none of the pixels in the lowest line of text are rendered because the line's raster position is invalid.

Consider how this can surprise you. Say you wanted to draw a string of text with each character rendered with glBitmap. Figure 3 shows a few situations. The point to notice is that the text renders as expected in the first two cases, but in the last case, the raster position's placement is outside the view frustum so no pixels from the last text string are drawn.

It would appear that there is no way to begin rendering of a string of text outside the bounds of the viewport and view frustum and render at least the ending portion of the string. There is a way to accomplish what you want; it is just not very obvious. The glBitmap command both draws a bitmap and then offsets the raster position in *relative* window coordinates. You can render the final line of text if you first position the raster position within the view frustum (so that the raster position is set valid), and then you offset the raster position by calling glBitmap with relative raster position offsets. In this case, be sure to specify a zero-width and zero-height bitmap so no pixels are actually rendered.

Here is an example of this:

```
glRasterPos2i(0, 0);
glBitmap(0, 0, 0, 0, xoffset, yoffset, NULL);
drawString("Will not be clipped.");
```

This code fragment assumes that the glRasterPos2i call will validate the raster position at the origin. The code to setup the projection and modelview matrix to do that is not show (setting both matrices to the identity matrix would be sufficient).

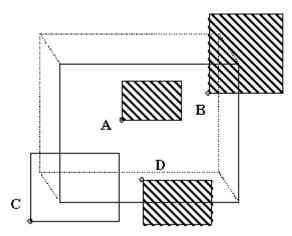


Figure 4: Various raster position scenarios. A, raster position is within the view frustum and the image is totally with the viewport. B, raster position is within the view frustum but the image is only partially within the viewport; still fragments are generated outside the viewport. C, raster position is invalid (due to being placed outside the view frustum); no pixels are rasterized. D, like case B except glpixelZoom(1, -1) has inverted the Y pixel rasterization direction so the image renders top to bottom.

10. The Viewport Does Not Clip or Scissor

It is a very common misconception that pixels cannot be rendered outside the OpenGL viewport. The viewport is often mistaken for a type of scissor. In fact, the viewport simply defines a transformation from normalized device coordinates (that is, post-projection matrix coordinates with the perspective divide applied) to window coordinates. The OpenGL specification makes no mention of clipping or culling when describing the operation of OpenGL âs viewport.

Part of the confusion comes from the fact that, most of the time, the viewport is set to be the windowâs rectangular extent and pixels are clipped to the windowâs rectangular extent. But do not confuse window ownership clipping with anything the viewport is doing because the viewport *does not* clip pixels.

Another reason that it seems like primitives are clipped by the viewport is that vertices are indeed clipped against the view frustum. OpenGLâs view frustum clipping does guarantee that no vertex (whether belonging to a geometric primitive or the raster position) can fall outside the viewport.

So if vertices cannot fall outside the view frustum and hence cannot be outside the viewport, how do pixels get rendered outside the viewport? Might it be an idle statement to say that the viewport does not act as a scissor if indeed you cannot generate pixels outside the viewport? Well, you *can* generate fragments that fall outside the viewport rectangle so it is not an idle statement.

The last section has already hinted at one way. While the raster position vertex must be specified to be within the view frustum to validate the raster position, once valid, the raster position (the state of which is maintained in window coordinates) can be moved outside the viewport with the glBitmap callâs raster position offset capability. But you do not even have to move the raster position outside the viewport to update pixels outside of the viewport rectangle. You can just render a large enough bitmap or image so that the pixel rectangle exceeds the extent of the viewport rectangle. Figure 4 demonstrates image rendering outside the viewport.

The other case where fragments can be generated outside the viewport is when rasterizing wide lines and points or smooth points, lines, and polygons. While the actual vertices for wide and smooth primitives will be clipped to fall within the viewport during transformation, at rasterization time, the widened rasterization footprint of wide or smooth primitives may end up generating fragments outside the boundaries of the viewport rectangle.

Indeed, this can turn into a programming pitfall. Say your application renders a set of wide points that slowly wander around on the screen. Your program configures OpenGL like this:

```
glViewport(0, 0, windowWidth, windowHeight);
glLineWidth(8.0);
```

What happens when a point slowly slides off the edge of the window? If the viewport matches the windowâs extents as indicated by the glviewport call above, you will notice that a point will disappear suddenly at the moment its center is outside the window extent. If you expected the wide point to gradually slide of the screen, that is not what happens!

Keep in mind that the extra pixels around a wide or antialiased point are generated at rasterization time, but if the pointâs vertex (at its center) is culled during vertex transformation time due to view frustum clipping, the widened rasterization never happens. You can fix the problem by widening the viewport to reflect the fact that a pointâs edge can be up to four pixels (half of 8.0) from the pointâs center and still generate fragments within the window âs extent. Change the glviewport call to:

glViewport(-4, -4, windowWidth+4, windowHeight+4);

With this new viewport, wide points can still be rasterized even if the hang off the window edge. Note that this will also slightly narrow your rectangular region of view, so if you want the identical view as before, you need to compensate by also expanding the view frustum specified by the projection matrix.

Note that if you really do require a rectangular 2D scissor in your application, OpenGL does provide a true window space scissor. See glenable(GL_SCISSOR_TEST) and glscissor.

10. Setting the Raster Color

Before you specify a vertex, you first specify the normal, texture coordinate, material, and color and then only when glVertex3f (or its ilk) is called will a vertex actually be generated based on the current per-vertex state. Calling glColor3f just sets the current color state. glColor3f does not actually create a vertex or any perform any rendering. The glVertex3f call is what binds up all the current per-vertex state and issues a complete vertex for transformation.

The raster position is updated similarly. Only when glRasterPos3f (or its ilk) is called does all the current per-vertex state get transformed and assigned to the raster position.

A common pitfall is attempting to draw a string of text with a series of glBitmap calls where different characters in the string are different colors. For example:

glColor3f(1.0, 0.0, 0.0); /* RED */
glRasterPos2i(20, 15);
glBitmap(w, h, 0, 0, xmove, ymove, red_bitmap);

glColor3f(0.0, 1.0, 0.0); /* GREEN */
glBitmap(w, h, 0, 0, xmove, ymove, green_bitmap);
/* WARNING: Both bitmaps render red. */

Unfortunately, glBitmap as relative offset of the raster position just updates the raster position location. The raster color (and the other remaining raster state values) remain unchanged.

The designers of OpenGL intentionally specified that glBitmap should not latch into place the current per-vertex state when the raster position is repositioned by glBitmap. Repeated glBitmap calls are designed for efficient text rendering with mono-chromatic text being the most common case. Extra processing to update per-vertex state would slow down the intended most common usage for glBitmap.

If you do want to switch the color of bitmaps rendered with glBitmap, you will need to explicitly call glRasterPos3f (or its ilk) to lock in a changed current color.

12. OpenGL's Lower Left Origin

Given a sheet of paper, people write from the top of the page to the bottom. The origin for writing text is at the upper lefthand margin of the page (at least in European languages). However, if you were to ask any decent math student to plot a few points on an X-Y graph, the origin would certainly be at the lower left-hand corner of the graph. Most 2D rendering APIs mimic writers and use a 2D coordinate system where the origin is in the upper left-hand corner of the screen or window (at least by default). On the other hand, 3D rendering APIs adopt the mathematically minded convention and assume a lower left-hand origin for their 3D coordinate systems.

If you are used to 2D graphics APIs, this difference of origin location can trip you up. When you specify 2D coordinates in OpenGL, they are generally based on a lower left-hand coordinate system. Keep this in mind when using glViewport, glScissor, glRasterPos2i, glBitmap, glTexCoord2f, glReadPixels, glCopyPixels, glCopyTexImage2D, glCopyTexSubImage2D, gluOrtho2D, and related routines.

Another common pitfall related to 2D rendering APIs having an upper left-hand coordinate system is that 2D image file formats start the image at the top scan line, not the bottom scan line. OpenGL assumes images start at the bottom scan line by default. If you do need to flip an image when rendering, you can use glpixelzoom(1, -1) to flip the image in the

Y direction. Note that you can also flip the image in the X direction. Figure 4 demonstrates using glPixelZoom to flip an image.

Note that glPixelZoom only works when rasterizing image rectangles with glDrawPixels or glCopyPixels. It does not work with glBitmap or glReadPixels. Unfortunately, OpenGL does not provide an efficient way to read an image from the frame buffer into memory starting with the top scan line.

13. Setting Your Raster Position to a Pixel Location

A common task in OpenGL programming is to render in window coordinates. This is often needed when overlaying text or blitting images onto precise screen locations. Often having a 2D window coordinate system with an upper left-hand origin matching the window systemâs default 2D coordinate system is useful.

Here is code to configure OpenGL for a 2D window coordinate system with an upper left-hand origin where $_{w}$ and $_{h}$ are the windowâs width and height in pixels:

```
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, w, h, 0, -1, 1);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Note that the bottom and top parameters (the 3rd and 4th parameters) to glortho specify the window height as the *top* and zero as the *bottom*. This flips the origin to put the origin at the windowâs upper left-hand corner.

Now, you can safely set the raster position at a pixel position in window coordinates like this

```
glVertex2i(x, y);
glRasterPos2i(x, y);
```

One pitfall associated with setting up window coordinates is that switching to window coordinates involves loading both the modelview and projection matrices. If you need to "get back" to what was there before, use glPushMatrix and glPopMatrix (but remember the pitfall about assuming the projection matrix stack has more than two entries).

All this matrix manipulation can be a lot of work just to do something like place the raster position at some window coordinate. Brian Paul has implemented a freeware version of the OpenGL API called Mesa. Mesa implements an OpenGL extension called MESA_window_pos that permits direct efficient setting of the raster position without disturbing any other OpenGL state. The calls are:

```
glWindowPos4fMESA(x,y,z,w);
glWindowPos2fMESA(x,y)
```

Here is the equivalent implementation of these routines in unextended OpenGL:

```
void
glWindowPos4fMESAemulate(GLfloat x,GLfloat y,GLfloat z,GLfloat w)
{
  GLfloat fx, fy;
  /* Push current matrix mode and viewport attributes. */
  glPushAttrib(GL_TRANSFORM_BIT | GL_VIEWPORT_BIT);
    /* Setup projection parameters. */
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
      glLoadIdentity();
      glMatrixMode(GL_MODELVIEW);
      glPushMatrix();
        glLoadIdentity();
        glDepthRange(z, z);
        glViewport((int) x - 1, (int) y - 1, 2, 2);
        /* Set the raster (window) position. */
        fx = x - (int) x;
```

```
fy = y - (int) y;
glRasterPos4f(fx, fy, 0.0, w);
/* Restore matrices, viewport and matrix mode. */
glPopMatrix();
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glPopAttrib();
}
void
glWindowPos2fMESAemulate(GLfloat x, GLfloat y)
{
glWindowPos4fMESAemulate(x, y, 0, 1);
}
```

Note all the extra work the emulation routines go through to ensure that no OpenGL state is disturbed in the process of setting the raster position. Perhaps commercial OpenGL vendors will consider implementing this extension.

14. Careful Enabling Color Material

OpenGL's color material feature provides a less expensive way to change material parameters. With color material enabled, material colors track the current color. This means that instead of using the relatively expensive glMaterialfv routine, you can use the glColor3f routine.

Here is an example using the color material feature to change the diffuse color for each vertex of a triangle:

```
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
glBegin(GL_TRIANGLES);
glColor3f(0.2, 0.5, 0.8);
glVertex3f(1.0, 0.0, 0.0);
glColor3f(0.3, 0.5, 0.6);
glVertex3f(0.0, 0.0, 0.0);
glColor3f(0.4, 0.2, 0.2);
glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

Consider the more expensive code sequence needed if glMaterialfv is used explicitly:

```
GLfloat d1 = { 0.2, 0.5, 0.8, 1.0 };
GLfloat d2 = { 0.3, 0.5, 0.6, 1.0 };
GLfloat d3 = { 0.4, 0.2, 0.2, 1.0 };
glBegin(GL_TRIANGLES);
glMaterialfv(GL_FRONT,GL_DIFFUSE,d1);
glWertex3f(1.0, 0.0, 0.0);
glMaterialfv(GL_FRONT,GL_DIFFUSE,d2);
glWertex3f(0.0, 0.0, 0.0);
glMaterialfv(GL_FRONT,GL_DIFFUSE,d3);
glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

If you are rendering objects that require frequent simple material changes, try to use the color material mode. However, there is a common pitfall encountered when enabling the color material mode. When color material is enabled, OpenGL immediately changes the material colors controlled by the color material state. Consider the following piece of code to initialize a newly create OpenGL rendering context:

```
GLfloat a[] = { 0.1, 0.1, 0.1, 1.0 };
glColor4f(1.0, 1.0, 1.0, 1.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, a);
glEnable(GL_COLOR_MATERIAL);
/* WARNING: Ambient and diffuse material latch immediately to the current
color. */
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColor3f(0.3, 0.5, 0.6);
```

What state will the front ambient and diffuse material colors be after executing the above code fragment? While the programmer may have intended the ambient material state to be (0.1, 0.1, 0.1, 1.0) and the diffuse material state to be (0.3, 0.5, 0.6, 1.0), that is not quite what happens.

The resulting diffuse material state is what the programmer intended, but the resulting ambient material state is rather unexpectedly (1.0, 1.0, 1.0, 1.0). How did that happen? Well, remember that the color material mode *immediately* begins tracking the current color when enabled. The initial value for the color material settings is GL_FRONT_AND_BACK and GL_AMBIENT_AND_DIFFUSE (probably not what you expected!).

Since enabling the color material mode immediately begins tracking the current color, both the ambient and diffuse material states are updated to be (1.0, 1.0, 1.0, 1.0). Note that the effect of the initial glMaterialfv is lost. Next, the color material state is updated to just change the front diffuse material. Lastly, the glColor3f invocation changes the diffuse material to (0.3, 0.5, 0.6, 1.0). The ambient material state ends up being (1.0, 1.0, 1.0, 1.0).

The problem in the code fragment above is that the color material mode is enabled before calling glColorMaterial. The color material mode is very effective for efficient simple material changes, but to avoid the above pitfall, always be careful to set glColorMaterial before you enable GL_COLOR_MATERIAL.

15. Much OpenGL State Affects All Primitives

A fragment is OpenGLâs term for the bundle of state used to update a given pixel on the screen. When a primitive such as a polygon or image rectangle is rasterized, the result is a set of fragments that are used to update the pixels that the primitive covers. Keep in mind that all OpenGL rendering operations share the same set of per-fragment operations. The same applies to OpenGLâs fog and texturing rasterization state.

For example, if you enabled depth testing and blending when you render polygons in your application, keep in mind that when you overlay some 2D text indicating the applicationâs status that you probably want to disable depth testing and blending. It is easy to forget that this state also affects images drawn and copied with glDrawPixels and glCopyPixels.

You will quickly notice when this shared state screws up your rendering, but also be aware that sometimes you can leave a mode enabled such as blending without noticing the extra expense involved. If you draw primitives with a constant alpha of 1.0, you may not notice that the blending is occurring and simply slowing you down.

This issue is not unique to the per-fragment and rasterization state. The pixel path state is shared by the draw pixels (glDrawPixels), read pixels (glReadPixels), copy pixels (glCopyPixels), and texture download (glTexImage2D) paths. If you are not careful, it is easy to get into situations where a texture download is screwed up because the pixel path was left configured for a pixel read back.

16. Be Sure to Allocate Ancillary Buffers that You Use

If you intend to use an ancillary buffer such as a depth, stencil, or accumulation buffer, be sure that you application actually requests all the ancillary buffers that you intend to use. A common interoperability issue is developing an OpenGL application on a system with only a few frame buffer configurations that provide all the ancillary buffers that you use. For example, your system has no frame buffer configuration that advertises a depth buffer without a stencil buffer. So on your development system, you "get away with" not explicitly requesting a stencil buffer.

The problem comes when you take your supposedly debugged application and run it on a new fancy hardware accelerated OpenGL system only to find out that the application fails miserably when attempting to use the stencil buffer. Consider that the fancy hardware may support extra color resolution if you do not request a stencil buffer. If you application does not explicitly request the stencil buffer that it uses, the fancy hardware accelerated OpenGL implementation determines that the frame buffer configuration with no stencil but extra color resolution is the better choice for your application. If your application would have correctly requested a stencil buffer things would be fine. Make sure that you allocate what you use.

Conclusion

I hope that this review of various OpenGL pitfalls saves you much time and debugging grief. I wish that I could have simply read about these pitfalls instead of learning most of them the hard way.

Visualization has always been the key to enlightenment. If computer graphics changes the world for the better, the fundamental reason why is that computer graphics makes visualization easier.

Table of Contents

OpenGL FAQ and Troubleshooting Guide v1.2000.10.15	1
1 About the FAQ	13
<u>2 Getting Started</u>	17
<u>3 GLUT</u>	32
<u>4 GLU</u>	35
<u>5 Microsoft Windows Specifics</u>	
6 Windows, Buffers, and Rendering Contexts	45
7 Interacting with the Window System, Operating System, and Input Devices	46
8 Using Viewing and Camera Transforms, and gluLookAt()	48
<u>9 Transformations</u>	52
10 Clipping, Culling, and Visibility Testing	61
<u>11 Color</u>	65
<u>12 The Depth Buffer</u>	67
13 Drawing Lines over Polygons and Using Polygon Offset	71
<u>14 Rasterization and Operations on the Framebuffer</u>	74
15 Transparency, Translucency, and Blending	79
16 Display Lists and Vertex Arrays	82
<u>17 Using Fonts</u>	85
18 Lights and Shadows	87
19 Curves, Surfaces, and Using Evaluators	92
20 Picking and Using Selection	93
21 Texture Mapping	96
22 Performance	
23 Extensions and Versions	

Table of Contents

24 Miscellaneous	
Appendix A Microsoft OpenGL Information	
Windows Driver Development Kits	
Preliminary Windows 2000 DDK	
Windows Driver and Hardware Development	
Fluff articles.	
MSDN Library	
Platform SDK	
OpenGL technical articles	
Useful other articles	
Knowledge Base.	
Current.	
Archive	
Appendix B Source Code Index	

OpenGL FAQ and Troubleshooting Guide v1.2000.10.15

<u>I About the FAQ</u>	15 Transparency, Translucency, and Blending
2 Getting Started	16 Display Lists and Vertex Arrays
<u>3 GLUT</u>	17 Using Fonts
<u>4 GLU</u>	18 Lights and Shadows
5 Microsoft Windows Specifics	19 Curves, Surfaces, and Using Evaluators
6 Windows, Buffers, and Rendering Contexts	20 Picking and Using Selection
7 Interacting with the Window System, Operating System, and Input Devices	21 <u>Texture Mapping</u>
8 Using Viewing and Camera Transforms, and gluLookAt()	22 Performance
9 <u>Transformations</u>	23 Extensions and Versions
10 Clipping, Culling, and Visibility Testing	24 Miscellaneous
11 <u>Color</u>	Appendix A Microsoft OpenGL Information
12 The Depth Buffer	Appendix B Source code index
13 Drawing Lines over Polygons and Using Polygon Offset	German Translation: OpenGL häufig gestellte fragen
14 Rasterization and Operations on the Framebuffer	<u>Japanese</u> <u>Translation:</u> OpenGL に関する質問と回答集

1 About the FAQ

- 1.010 Introduction
- 1.020 How to contribute, and the contributors
- 1.030 Download the entire FAQ as a Zip file
- 1.031 Printing the PDF FAQ
- 1.040 Change Log

2 Getting Started

- 2.005 Where can I find 3D graphics info?
- 2.010 Where can I find examples, tutorials, documentation, and other OpenGL information?
- 2.020 What OpenGL books are available?
- 2.030 What OpenGL chat rooms and newsgroups are available?
- 2.040 What OpenGL implementations come with source code?
- 2.050 What compiler can I use?

2.060 What do I need to compile and run OpenGL programs?

2.070 Why am I getting compile, link, and runtime errors?

2.080 How do I initialize my windows, create contexts, etc.?

2.090 How do I create a full-screen window?

2.100 What is the general form of an OpenGL program?

2.110 My window is blank. What should I do?

2.120 My first frame renders correctly, but subsequent frames are incorrect or further away or I just get a blank screen. What's going on?

2.130 What is the AUX library?

2.140 What support for OpenGL does {Open,Net,Free}BSD or Linux provide?

2.150 Where is OpenGL 1.2?

2.160 What are the OpenGL Conformance Tests?

3 <u>GLUT</u>

3.010 What is GLUT? How is it different from OpenGL?

3.020 Should I use GLUT?

3.030 I need to set up different tasks for left and right mouse button motion. However, I can only set one glutMotionFunc() callback, which doesn't pass the button as a parameter.

3.040 How does GLUT do...?

3.050 How can I perform animations with GLUT?

3.060 Is it possible to change a window's size *after* it's opened (i.e., after I call glutInitWindowSize(); and glutCreateWindow();)?

3.070 <u>I have a GLUT program that allocates memory at startup. How do I deallocate this memory when the program exits?</u>

3.080 How can I make my GLUT program detect that the user has closed the window?

3.090 How can I make glutMainLoop() return to my calling program?

3.100 How do I get rid of the console window in a Windows GLUT application?

3.110 My GLUT question isn't answered here. Where can I get more info?

4 <u>GLU</u>

OpenGL FAQ and Troubleshooting Guide v1.2000.10.15

- 4.010 What is GLU? How is it different from OpenGL?
- 4.020 How does GLU render sphere, cylinder, and disk primitives?
- 4.030 How does gluPickMatrix work?
- 4.040 How do I use GLU tessellation routines?
- 4.050 Why aren't my tessellation callback routines called?
- 4.060 How do I use GLU NURBS routines?
- 4.070 How do I use gluProject and gluUnProject?

5 Microsoft Windows Specifics

- 5.010 What's a good source for Win32 OpenGL programming information?
- 5.020 I'm looking for a Wintel OpenGL card in a specific price range, any suggestions?
- 5.030 How do I enable and disable hardware rendering on a Wintel card?
- 5.040 How do I know my program is using hardware acceleration on a Wintel card?
- 5.050 Where can I get the OpenGL ICD for a Wintel card?
- 5.060 I'm using a Wintel card and an OpenGL feature doesn't seem to work. What's going on?
- 5.070 Can I use OpenGL with DirectDraw?
- 5.080 Is it ok to use DirectDraw to change the screen resolution or desktop pixel depth?

5.090 My card supports OpenGL, but I don't get acceleration regardless of which pixel format I try.

- 5.100 How do I get hardware acceleration?
- 5.110 Why doesn't OpenGL hardware acceleration work with multiple monitors?
- 5.120 Why does my MFC window flash, even though I'm using d
- 5.121 Why does my double buffered window appear incomplete or contain black stripes?
- 5.130 What's the difference between opengl.dll and opengl32.dll?
- 5.140 Should I use Direct3D or OpenGL?
- 5.150 What do I need to know to use OpenGL with MFC?
- 5.160 How can I use OpenGL with MFC?

- 5.170 Is OpenGL inherently slower when used with MFC?
- 5.180 Where can I find MFC examples?
- 5.190 What do I need to know about mixing WGL and GDI calls?
- 5.200 Why does my code crash under Windows NT or 2000 but run fine under 9x?
- 5.210 How do I properly use WGL functions?
- 6 <u>Windows, Buffers, and Rendering Contexts</u>
 - 6.010 How do I use overlay planes?
- 7 Interacting with the Window System, Operating System, and Input Devices
 - 7.010 How do I obtain the window width and height or screen max width and height?
 - 7.020 What user interface system should I use?
 - 7.030 How can I use multiple monitors?

8 Using Viewing and Camera Transforms, and gluLookAt()

- 8.010 How does the camera work in OpenGL?
- 8.020 How can I move my eye, or camera, in my scene?
- 8.030 Where should my camera go, the ModelView or projection matrix?
- 8.040 How do I implement a zoom operation?
- 8.050 <u>Given the current ModelView matrix, how can I determine the object–space location</u> <u>of the camera?</u>
- 8.060 How do I make the camera "orbit" around a point in my scene?
- 8.070 <u>How can I automatically calculate a view that displays my entire model? I know the bounding sphere and up vector.</u>
- 8.080 Why doesn't gluLookAt work?
- 8.090 How do I get a specified point (XYZ) to appear at the center of the scene?
- 8.100 I put my gluLookAt() call on my Projection matrix and now fog, lighting, and texture mapping don't work correctly. What happened?
- 8.110 How can I create a stereo view?

9 Transformations

9.001 I can't get transformations to work. Where can I learn more about matrices?

9.005 <u>Are OpenGL matrices column-major or row-major?</u>

9.010 What are OpenGL coordinate units?

9.011 How are coordinates transformed? What are the different coordinate spaces?

9.020 How do I transform only one object in my scene or give each object its own transform?

9.030 How do I draw 2D controls over my 3D rendering?

9.040 How do I bypass OpenGL matrix transformations and send 2D coordinates directly for rasterization?

9.050 What are the pros and cons of using absolute versus relative coordinates?

9.060 How can I draw more than one view of the same scene?

9.070 How do I transform my objects around a fixed coordinate system rather than the object's local coordinate system?

9.080 What are the pros and cons of using glFrustum() versus gluPerspective()? Why would I want to use one over the other?

9.085 How can I make a call to glFrustum() that matches my call to gluPerspective()?

9.090 How do I draw a full-screen quad?

9.100 How can I find the screen coordinates for a given object-space coordinate?

9.110 How can I find the object-space coordinates for a pixel on the screen?

9.120 How do I find the coordinates of a vertex transformed only by the ModelView matrix?

9.130 How do I calculate the object-space distance from the viewer to a given point?

9.140 How do I keep my aspect ratio correct after a window resize?

9.150 Can I make OpenGL use a left-handed coordinate space?

9.160 How can I transform an object so that it points at or follows another object or point in my scene?

9.162 How can I transform an object with a given yaw, pitch, and roll?

9.170 How can I render a mirror?

9.180 How can I do my own perspective scaling?

10 Clipping, Culling, and Visibility Testing

OpenGL FAQ and Troubleshooting Guide v1.2000.10.15

10.010 How do I tell if a vertex has been clipped or not?

10.020 How do I perform occlusion or visibility testing?

10.030 How do I render to a nonrectangular viewport?

10.040 When an OpenGL primitive moves placing one vertex outside the window, suddenly the color or texture mapping is incorrect. What's going on?

10.050 <u>I know my geometry is inside the view volume. How can I turn off OpenGL's view-volume clipping to maximize performance?</u>

10.060 When I move the viewpoint close to an object, it starts to disappear. How can I disable OpenGL's zNear clipping plane?

10.070 <u>How do I draw glBitmap or glDrawPixels primitives that have an initial glRasterPos</u> outside the window's left or bottom edge?

10.080 Why doesn't glClear work for areas outside the scissor rectangle?

10.090 How does face culling work? Why doesn't it use the surface normal?

11 <u>Color</u>

11.010 My texture map colors reverse blue and red, yellow and cyan, etc. What's going on?

11.020 How do I render a color index into an RGB window or vice versa?

11.030 The colors are almost entirely missing when I render in Microsoft Windows. What's happening?

11.040 How do I specify an exact color for a primitive?

11.050 How do I render each primitive in a unique color?

12 <u>The Depth Buffer</u>

12.010 How do I make depth buffering work?

12.020 Depth buffering doesn't work in my perspective rendering. What's going on?

12.030 How do I write a previously stored depth image to the depth buffer?

12.040 Depth buffering seems to work, but polygons seem to bleed through polygons that are in front of them. What's going on?

12.050 Why is my depth buffer precision so poor?

12.060 How do I turn off the zNear clipping plane?

12.070 Why is there more precision at the front of the depth buffer?

12.080 There is no way that a standard—sized depth buffer will have enough precision for my astronomically large scene. What are my options?

13 Drawing Lines over Polygons and Using Polygon Offset

13.010 What are the basics for using polygon offset?

13.020 What are the two parameters in a glPolygonOffset() call and what do they mean?

13.030 <u>What's the difference between the OpenGL 1.0 polygon–offset extension and OpenGL 1.1 (and later) polygon–offset interfaces?</u>

13.040 Why doesn't polygon offset work when I draw line primitives over filled primitives?

13.050 What other options do I have for drawing coplanar primitives when I don't want to use polygon offset?

14 Rasterization and Operations on the Framebuffer

14.010 <u>How do I obtain the address of the OpenGL framebuffer, so that I might write directly to it?</u>

14.015 <u>How do I use glDrawPixels() and glReadPixels()?</u>

14.020 How do I change between double- and single-buffered mode in an existing window?

14.030 How do I read back a single pixel?

14.040 How do I obtain the Z value for a rendered primitive?

14.050 How do I draw a pattern into the stencil buffer?

14.060 How do I copy from the front buffer to the back buffer and vice versa?

14.070 Why don't I get valid pixel data for an overlapped area, when I call glReadPixels where part of the window is overlapped by another window?

14.080 Why does the appearance of my smooth-shaded quad change when I view it with different transformations?

14.090 How do I obtain exact pixelization of lines?

14.100 How do I turn on wide-line endpoint capping or mitering?

14.110 How do I render rubber band lines?

14.120 If I draw a quad in fill mode and again in line mode, why don't the lines hit the same pixels as the filled quad?

14.130 How do I draw a full-screen quad?

14.140 How do I initialize or clear a buffer without calling glClear()?

14.150 How can I make line or polygon antialiasing work?

14.160 How do I achieve full-scene antialiasing?

15 Transparency, Translucency, and Using Blending

15.010 What is the difference between transparent, translucent, and blended primitives?

15.020 How can I achieve a transparent effect?

15.030 How can I create screen door transparency?

15.040 How can I render glass with OpenGL?

15.050 Do I need to render my primitives from back to front for correct rendering of translucent primitives to occur?

15.060 <u>I want to use blending but can't get destination alpha to work. Can I blend or create a transparency effect without destination alpha?</u>

15.070 If I draw a translucent primitive and draw another primitive behind it, I expect the second primitive to show through the first, but it's not there at all. Why not?

15.080 How can I make part of my texture maps transparent or translucent?

16 Display Lists and Vertex Arrays

16.010 Why does a display list take up so much memory?

16.020 How can I share display lists between different contexts?

16.030 How does display list nesting work? Is the called list copied into the calling list?

16.040 How can I do a particular function while a display list is called?

16.050 <u>How can I change an OpenGL function call in a display list that contains many other</u> <u>OpenGL function calls?</u>

16.060 <u>How can I obtain a list of function calls and the OpenGL call parameters from a display list?</u>

16.070 I've converted my program to use display lists, and it doesn't run any faster! Why not?

16.080 To save space, should I convert all my coordinates to short before storing them in a display list?

16.090 Will putting textures in a display list make them run faster?

16.100 Will putting vertex arrays in a display list make them run faster?

16.110 When sharing display lists between contexts, what happens when I delete a display list in one context? Do I have to delete it in all the contexts to make it really go away?

- 16.120 How many display lists can I create?
- 16.130 How much memory does a display list use?
- 16.140 How will I know if the memory a display list uses is freed?
- 16.150 How can I use vertex arrays to share vertices?

17 Using Fonts

- 17.010 How can I add fonts to my OpenGL scene?
- 17.020 How can I use TrueType fonts in my OpenGL scene?
- 17.030 How can I make 3D letters that I can light, shade, and rotate?

18 Lights and Shadows

- 18.010 What should I know about lighting in general?
- 18.020 Why are my objects all one flat color and not shaded and illuminated?
- 18.030 How can I make OpenGL automatically calculate surface normals?
- 18.040 Why can I only get flat shading when I light my model?
- 18.050 How can I make my light move or not move and control the light position?
- 18.060 How can I make a spotlight work?
- 18.070 How can I create more lights than GL MAX LIGHTS?
- 18.080 Which is faster: making glMaterial*() calls or using glColorMaterial()?
- 18.090 Why is the lighting incorrect after I scale my scene to change its size?
- 18.100 After I turn on lighting, everything is lit. How can I light only some of the objects?
- 18.110 How can I use light maps (e.g., Quake-style) in OpenGL?
- 18.120 How can I achieve a refraction lighting effect?
- 18.130 <u>How can I render caustics?</u>
- 18.140 How can I add shadows to my scene?

19 Curves, Surfaces, and Using Evaluators

19.010 How can I use OpenGL evaluators to create a B-spline surface?

19.020 How can I retrieve the geometry values produced by evaluators?

20 Picking and Using Selection

20.010 How can I know which primitive a user has selected with the mouse?

20.020 What do I need to know to use selection?

20.030 Why doesn't selection work?

20.040 How can I debug my picking code?

20.050 How can I perform pick highlighting the way PHIGS and PEX provided?

21 <u>Texture Mapping</u>

21.010 What are the basic steps for performing texture mapping?

21.020 I'm trying to use texture mapping, but it doesn't work. What's wrong?

21.030 Why doesn't lighting work when I turn on texture mapping?

21.040 Lighting and texture mapping work pretty well, but why don't I see specular highlighting?

- 21.050 How can I automatically generate texture coordinates?
- 21.060 Should I store texture maps in display lists?
- 21.070 How do texture objects work?
- 21.080 Can I share textures between different rendering contexts?
- 21.090 How can I apply multiple textures to a surface?
- 21.100 How can I perform light mapping?
- 21.110 How can I turn my files, such as GIF, JPG, BMP, etc. into a texture map?
- 21.120 How can I render into a texture map?
- 21.130 What's the maximum size texture map my device will render hardware accelerated?
- 21.140 How can I texture map a sphere, cylinder, or any other object with multiple facets?

22 <u>Performance</u>

22.010 What do I need to know about performance?

22.020 How can I measure my application's performance?

22.030 Which primitive type is the fastest?

22.040 What's the cost of redundant calls?

22.050 <u>I have (n) lights on, and when I turned on (n+1), suddenly performance dramatically</u> <u>dropped. What happened?</u>

22.060 I'm using (n) different texture maps and when I started using (n+1) instead, performance drastically dropped. What happened?

22.070 Why are glDrawPixels() and glReadPixels() so slow?

22.080 Is it faster to use absolute coordinates or to use relative coordinates?

22.090 Are display lists or vertex arrays faster?

22.100 How do I make triangle strips out of triangles?

23 Extensions and Versions

23.010 Where can I find information on different OpenGL extensions?

23.020 How will I know which OpenGL version my program is using?

23.030 What is the difference between OpenGL 1.0, 1.1, and 1.2?

23.040 How can I code for different versions of OpenGL?

23.050 How can I find which extensions are supported?

23.060 How can I code for extensions that may not exist on a target platform?

23.070 How can I call extension routines on Microsoft Windows?

23.080 How can I call extension routines on Linux?

23.090 Where can I find extension enumerants and function prototypes?

24 Miscellaneous

24.010 How can I render a wireframe scene with hidden lines removed?

24.020 How can I render rubber-band lines?

24.030 <u>My init code calls glGetString() to find information about the OpenGL</u> implementation, but why doesn't it return a string?

24.039 Where can I find 3D model files?

24.040 How can I load geometry files, such as 3DS, OBJ, DEM, etc. and render them with OpenGL?

24.050 How can I save my OpenGL rendering as an image file, such as GIF, TIF, JPG, BMP, etc.? How can I read these image files and use them as texture maps?

24.060 Can I use a BSP tree with OpenGL?

24.070 Can I use an octree with OpenGL?

24.080 Can I do radiosity with OpenGL?

24.090 Can I raytrace with OpenGL?

24.100 How can I perform CSG with OpenGL?

24.110 How can I perform collision detection with OpenGL?

24.120 <u>I understand OpenGL might cache commands in an internal buffer. Can I perform an abort operation, so these buffers are simply emptied instead of executed?</u>

24.130 What's the difference between glFlush() and glFinish() and why would I want to use these routines?

24.140 How can I print with OpenGL?

24.150 Can I capture or log the OpenGL calls an application makes?

24.160 How can I render red-blue stereo pairs?

Appendix A Microsoft OpenGL Information

Appendix B Source Code Index

1 About the FAQ

1.010 Introduction

The OpenGL Technical FAQ and Troubleshooting Guide will answer some basic technical questions and explain frequently misunderstood topics, features, and concepts.

All text, example code, and code snippets in this FAQ are in the public domain. The text, example code, and code snippets can be used and copied freely. Hyperlinks to text and example code not contained in this FAQ may or may not be public domain, and their usage may be restricted accordingly.

1.020 How to contribute, and the contributors

This FAQ is maintained by Paul Martz (paul martz@hp.com).

Contribute to the FAQ by contacting Paul Martz, the FAQ maintainer. Suggestions, topics, corrections, information, and pointers to information are welcome.

The following people have explicitly contributed written material to this FAQ: Brian Bailey, Brett Johnson, Paul Martz, Samuel Paik, Joel Parris, and Thant Tessman.

Several people have unwittingly contributed information through conversations with the FAQ maintainer and/or their several informative postings to the comp.graphics.api.opengl newsgroup. A partial list includes: Darren Adams, Stephane Albi, Mark B. Allan, Pierre Alliez, Steve Baker, Konstantin Baumann, Ron Bielaski, Kevin Bjorke, Lars Blaabjerg, Frans Bouma, Michael Brooks, Jeff Burrell, Won Chun, Mike Coplien, Bart De Lathouwer, Angus Dorbie, Bob Ellison, Glenn Forney, Ron Fosner, Phil Frisbie Jr, Michael I. Gold, Paul Groves, Charles E. Hardwidge, Jason Harrison, Michael S. Harrison, Mike Heck, Chris Hecker, Scott Heiman, Helios, Blaine Hodge, Steve Humphreys, Michael Kennedy, Marco Klemm, Mark Kilgard, Oliver Kurowski, Michael Kurth, Bruce Lamming, Robert Lansdale, Jon Leech, Stuart Levy, Barthold Lichtenbelt, Mike Lischke, Ben Loftin, Jean–Luc Martinez, Steve McAndrewSmith, Phil McRevis, David Melinosky, Reed Mideke, Teri Morrison, Duncan Murdoch, Doug Newlin, Geert Poels, David Poon, Lev Povalahev, Dirk Reiners, Stephane Routelous, Schneide, Shaleh, Dave Shreiner, Hal Snyder, Andrew F. Vesper, Jon White, Lucian Wischik, Mitch Wolberg, and Zed.

Jeff Molofee's OpenGL code was the inspiration for Brian Bailey's MFC example (accessible from question 5.160). Jeff maintains <u>the NeHe Web page</u>.

Special thanks to Yukio Andoh for the <u>Japanese translation</u>, and Thomas Kern for the <u>German translation</u>.

1.030 Download the entire FAQ as a Zip file

Download the entire FAQ in a single <u>zip file (~180KB)</u>.

1.031 Printing the PDF FAQ

The entire FAQ is available as a single PDF file for easy printing.

PDF FAQ (~610KB) Zipped PDF FAQ (~324KB)

1.040 Change Log

Date	Notes
October 15, 2000	Table of Contents: Version is now present in masthead.Table of Contents: Corrected Kanji characters.7.030: Added more information on multiple monitor support.17.010, 17.030: Repaired or removed broken links.
October 8, 2000	 source.htm: New file, a consolidated index to FAQ source code. Table of Contents: Added links to German and Japanese translations. Table of Contents: Added link to source.htm as Appendix B. 2.005: Fixed broken link. 2.010: Added information. 2.110: Added information. 5.030: Added information on disabling hardware rendering. 5.070: Added information. 5.080: Added information. 5.121: New question, "Why does my double buffered window appear incomplete or contain black stripes?" 5.160: Fixed HTML. 5.180: New question, "Where can I find MFC examples?" 5.190: New question, "Why does my code crash under Windows NT or 2000 but run fine under 9x?" 5.201: New question, "How do I properly use WGL functions?" 7.030: New question, "How can I use multiple monitors?" 8.110: New question, "How can I create a stereo view?" 9.005: Added information. 9.162: New question, "How can I transform an object with a given yaw, pitch, and roll?" 10.020: Added information. 14.160: Added information. 24.160: Added information.
August 24, 2000	Table of Contents: Added access to mslinks.htm as an appendix in the main table of contents 1.031: New question, "Printing the PDF FAQ" lookat.cpp: Fix comment typo.
August 1, 2000	mslinks.htm: New file, contains links to OpenGL information on Microsoft Web sites. 2.005: Added information. 2.010: Added link to mslinks.htm. 2.050: Added information.

	 2.080: Fixed incorrect parameters to XCreateWindow(). 2.160: New question, "What are the OpenGL Conformance Tests?" 3.020: Fixed typo. 5.010: Added link to mslinks.htm. 5.050: Added information. 5.150 New question, "What do I need to know to use OpenGL with MFC?" 5.160: New question, "How can I use OpenGL with MFC?" 5.170: New question, "Is OpenGL inherently slower when used with MFC?" 6.010: New question, "How do I use overlay planes?" 7.020: Added information. 9.001: Fixed hyperlink. 17.010: Removed broken hyperlink. 23.090: Added information. 24.050: Fixed hyperlink.
July 6, 2000	 2.005: Added information. 2.020: Added hyperlink to online OpenGL Reference Manual. 3.030: Corrected code snippet. 3.070: Added information. 3.090: Added information. 4.020: Added hyperlink to GLE web site. 5.040: Added information. 7.020 New question, "What user interface system should I use?" 9.011 New question, "How are coordinates transformed? What are the different coordinate spaces?" 16.150: New question, "How can I use vertex arrays to share vertices?" 17.010: Added information. 17.030: Added additional links to GLTT. 21.090: Added information. 21.110: Added link to source for using TGA files as texture maps. Corrected bogus hyperlink. 22.020: Added information. 22.100 New question, "How doI make triangle strips out of triangles?" 23.070: Added direct links to glext.h for hiding function pointers. 23.090: Added information. 24.050: Added information. 24.050: Added information.
June 6, 2000	Added a link to GLTT in question 17.030.
June 3, 2000	Updated with miscellaneous corrections and additional information.
May 30, 2000	Fixed HTML problems in index.htm, and sections 1 and 2.

May 29, 2000	Updated with miscellaneous corrections
May 28, 2000	Version 1.0. Significant changes include a full technical edit for hyperlinks, notational conventions, grammar, and spelling. Filled in many holes. Corrected lots of incorrect information.
April 16, 2000	Beta version.
March 19, 2000	Updated with miscellaneous corrections, additions, and changes.
March 12, 2000	Alpha version.

2 Getting Started

2.005 Where can I find 3D graphics info?

The <u>comp.graphics.algorithms FAQ</u> contains 3D graphics information that isn't specific to OpenGL.

For general OpenGL and 3D graphics information, <u>Advanced Graphics Programming</u> <u>Techniques Using OpenGL</u> is a good online source of information.

An excellent general computer graphics text is *Computer Graphics: Principles and Practice*, Second Edition, by James Foley, et al. ISBN 0–201–12110–7. This book may be out of print, however, some online book retailers still seem to have it for sale. Try <u>amazon.com</u>. There may be a third edition planned for release in January 2001

Delphi code for performing basic vector, matrix, and quaternion operations can be found here.

Here's another source for linear algebra source code.

2.010 Where can I find examples, tutorials, documentation, and other OpenGL information?

OpenGL is the most extensively documented 3D graphics API to date. Information is all over the Web and in print. It would be impossible to exhaustively list all sources of OpenGL information. This FAQ therefore provides links to large storehouses of information and sites that maintain many links to other OpenGL sites.

OpenGL Organization Web Page

SGI's OpenGL Web site and (apparently) SGI's other OpenGL Web site.

HP's OpenGL subject index.

OpenGL Basics FAQ

<u>OpenGL Game Developer's FAQ</u>. In addition to information on OpenGL, the OpenGL Game Developer's FAQ has information on subscribing to the OpenGL Game Developer's mailing list.

The EFnet #OpenGL FAQ

Samuel Paik has created a large repository of <u>links to OpenGL information on Microsoft</u> <u>Web sites</u>.

<u>The OpenGL org web site</u> has the current OpenGL specification and manual pages. You can view the OpenGL spec v1.1 <u>online as a Web page</u>.

A repository of OpenGL implementations for several platforms

<u>The GLUT source code distribution</u> contains several informative OpenGL examples and demos.

Codeguru maintains a small, but growing list, of useful OpenGL sample code.

Lucian Wischik's Web page at <u>http://www.wischik.com/lu/programmer/wingl.html</u> contains excellent information on Microsoft Windows OpenGL, especially with 3dfx hardware.

<u>The NeHe Web page</u> has many links to other sites and plenty of useful tutorials. Many people have found this site useful.

See Blaine Hodge's Web page for info on Win32 OpenGL programming.

An interactive **OpenGL** tutorial can be found here.

Check gamedev.net for OpenGL tutorials and articles.

2.020 What OpenGL books are available?

There are several books on OpenGL, but the two most revered are the "red" and "blue" books:

OpenGL Programming Guide, Third Edition, Mason Woo et al. ISBN 0–201–60458–2 (aka the red book)

OpenGL Reference Manual, Third Edition, Dave Shreiner (Editor), et al. ISBN 0–201–65765–1 (aka the blue book)

The third edition of these books describes OpenGL 1.2. The original and second editions describe 1.0 and 1.1, respectively.

The OpenGL red book is online.

For the OpenGL Reference Manual, here are two sources:

HP's Web–browsable <u>OpenGL Reference Manual, Second Edition (for OpenGL 1.1)</u>. Manual pages similar to the <u>OpenGL Reference Manual</u>.

In addition to the red and blue books, see the green book for X Windows programming, and the white book for Microsoft Windows programming. You can obtain a more exhaustive list of OpenGL books by visiting the <u>www.opengl.org</u> Web site.

2.030 What OpenGL chat rooms and newsgroups are available?

The Usenet newsgroup, devoted to OpenGL programming, is comp.graphics.api.opengl.

The #OpenGL IRC channel is devoted to OpenGL discussion.

2.040 What OpenGL implementations come with source code?

The <u>Mesa library</u> is an OpenGL look–alike. It has an identical interface to OpenGL. The only reason it can't be called "OpenGL" is because its creator hasn't purchased a license from the OpenGL ARB.

The **OpenGL Sample Implementation** is also available.

2.050 What compiler can I use?

OpenGL programs are typically written in C and C++. You can also program OpenGL from Delphi (a Pascal–like language), Basic, Fortran, Ada, and others.

Borland

Programming OpenGL with Borland compilers is the same as with any other compiler, with one exception: OpenGL apps can produce floating point exceptions at run time. To disable these harmless errors, add the following to your app before you call an OpenGL function:

_control87(MCW_EM, MCW_EM);

Borland users need to be aware that versions prior to 4.0 only support OpenGL 1.0 out of the box. Download the OpenGL SDK from Microsoft to use OpenGL v1.1, or v1.2 when it becomes available.

Use Borland's implibutility to generate Borland-compatible .LIB export libraries from Microsoft-compatible .DLL libraries. If you accidently link with Microsoft-format .LIB files, you will receive a linker error like the following:

C:\BORLAND\BCC55\LIB\GLUT32.LIB' contains invalis OMF record, type 0x21 (possibly COOF)

The bornews.borland.com Usenet news server has two newsgroups that pertain to graphics: borland.public.delphi.graphics and borland.public.cppbuilder.graphics.

The Borland Community is an online source of FAQs that address Borland compiler issues.

For information on how to use OpenGL through the commercial version of Borland C++ Builder, visit <u>Scott</u> <u>Heiman's Web page</u>. For information on the <u>free version, go here</u>.

The book *Delphi Developer's Guide to OpenGL* by Jon Jacobs is available. <u>The author maintains a web page</u> for this book.

<u>Information on using OpenGL from Delphi can be found here</u> and at the <u>Delphi3D web page</u>. Code and utilities for <u>using OpenGL through Delphi are available</u>.

Visual Basic

Here are three sites with info on how to use OpenGL through Visual Basic:

http://www.softoholic.bc.ca/opengl/down.htm http://www.weihenstephan.de/~syring/ActiveX/ http://www.ieighty.net/~davepamn/colorcube.html.

2.060 What do I need to compile and run OpenGL programs?

The following applies specifically to C/C++ usage.

To compile and link OpenGL programs, you'll need OpenGL header files and libraries. To

2 Getting Started

run OpenGL programs you may need shared or dynamically loaded OpenGL libraries, or a vendor–specific OpenGL Installable Client Driver (ICD) specific to your device. Also, you may need include files and libraries for the GLU and GLUT libraries. Where you get these files and libraries will depend on which OpenGL system platform you're using.

The OpenGL Organization maintains a list of <u>links to OpenGL developer and end–user files</u>. You can download most of what you need from there.

Under Microsoft Windows 9x, NT, and 2000:

If you're using Visual C++, your compiler comes with include files for OpenGL and GLU, as well as .lib files to link with.

For GLUT, download these files. Install glut.h in your compiler's include directory, glut32.lib in your compiler's lib directory, and glut32.dll in your Windows system directory (c:\windows\system for Windows 9x, or c:\winnt\system32 for Windows NT/2000).

In summary, a fully installed Windows OpenGL development environment will look like this:

File	Location
gl.h glut.h glu.h	[compiler]\include\gl
Opengl32.lib glut32.lib glu32.lib	[compiler]\lib
Opengl32.dll glut32.dll glu32.dll	[system]

where [compiler] is your compiler directory (such as c:\Program Files\Microsoft Visual Studio\VC98) and [system] is your Windows 9x/NT/2000 system directory (such as c:\winnt\system32 or c:\windows\system).

If you're on a hardware platform that accelerates OpenGL, you'll need to install the ICD for your device. This may have shipped with your hardware, or you can download it from your hardware vendor's Web page. Your vendor may also provide a replacement or addition for gl.h, which provides definitions and declarations for vendor–specific OpenGL extensions. See the extensions section in this FAQ for more information.

If you see files such as opengl.lib and glut.lib, these are SGI's unsupported libraries for Microsoft Windows. They should not be used. To use hardware acceleration, the Microsoft libraries are recommended. <u>More info on the SGI libraries can be found here</u>. Always link with either all Microsoft libraries (e.g., glu32.lib, glut32.lib, and opengl32.lib) or all SGI libraries (e.g., glu.lib, glut.lib, and opengl.lib). You can't use a combination of both Microsoft libraries and SGI libraries. However, you can install both sets of libraries on the same

system. If you use SGI's .lib files, you'll need the corresponding .dll files installed in your system folder. (i.e., linking against opengl.lib requires that opengl.dll is installed at run time).

You'll need to instruct your compiler to link with the OpenGL, GLU, and GLUT libraries. In Visual C++ 6.0, you can accomplish this with the Project menu's Settings dialog box. Scroll to the Link tab. In the Object/library modules edit box, add glut32.lib, glu32.lib, and opengl32.lib to the end of any text that is present.

For UNIX or UNIX-like operating systems:

If you don't find the header files and libraries that you need to use in standard locations, you need to point the compiler and linker to their location with the appropriate -I and -L options. The libraries you link with must be specified at link time with the -l option; -lglut -lGLU-lGL -lXmu -lX11 is typical.

If you want to use GLUT, you need to download it. If you can't find the precompiled binaries, you'll want to download the source and compile it. GLUT builds easily on many platforms, and comes with many README files explaining how to do a build. The GLUT compiler uses the imake utility, which makes it easy to build GLUT on new platforms.

For Linux, Macintosh, and other systems:

<u>Mesa</u> is a free OpenGL–like library that is available on a number of platforms. You might also check the Developer section at <u>The OpenGL Organization's Web page</u> for information about OpenGL for your specific platform.

2.070 Why am I getting compile, link, and runtime errors?

Most compile and link errors stem from either a system that doesn't have the OpenGL development environment installed correctly, or failure to instruct the compiler where to find the include and library files.

If you are encountering these problems in the Windows 9x/NT/2000 environment, read <u>question 2.060 above</u> to ensure that you've installed all files in their correct locations, and that you've correctly instructed the linker to find the .lib files.

Also, note that you'll need to put an #include <windows.h> statement before the #include<GL/gl.h>. Microsoft requires system DLLs to use a specific calling convention that isn't the default calling convention for most Win32 C compilers, so they've annotated the OpenGL calls in gl.h with some macros that expand to nonstandard C syntax. This causes Microsoft's C compilers to use the system calling convention. One of the include files included by windows.h defines the macros.

Another caveat for Win32 developers: With Microsoft Visual C++ (and probably most other Win32 C compilers), the standard Win32 application entry point is WinMain with four parameters, rather than main(int argc, char **argv). Visual C++ has an option to include code to parse the standard Win32 application entry, and call main with a parsed command line; this is called a console application instead of a Win32 application. If you download code from the Net and try to build it, make sure you've configured your compiler to build the right kind of application, either console or Win32. This can be controlled with linker options or pragmas. Microsoft Visual C++ supports the following pragmas for controlling the entry

point and application type:

```
// Use one of:
#pragma comment (linker, "/ENTRY:mainCRTStartup")
#pragma comment (linker, "/ENTRY:wmainCRTStartup")
#pragma comment (linker, "/ENTRY:WinMainCRTStartup")
#pragma comment (linker, "/ENTRY:wWinMainCRTStartup")
// Use one of:
#pragma comment (linker, "/SUBSYSTEM:WINDOWS")
#pragma comment (linker, "/SUBSYSTEM:CONSOLE")
```

The following is a table of errors and their possible causes and solutions. It is targeted toward
Microsoft Visual C++ users, but the types of errors can apply, in general, to any platform.

Example error text	Possible cause and solution
d:\c++\file.c(20) : warning C4013: 'glutDestroyWindow' undefined; assuming extern returning int d:\c++\file.c(71) : warning C4013: 'glMatrixMode' undefined; assuming extern returning int d:\c++\file.c(71) : error C2065: 'GL_MODELVIEW' : undeclared identifier	Didn't #include gl.h, glu.h, or glut.h A GLUT source file should: #include <gl glut.h=""> Non-GLUT source files should: #include <gl glu.h=""> #include <gl gl.h=""></gl></gl></gl>
c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1152) : error C2054: expected '(' to follow 'WINGDIAPI' c:\program files\microsoft visual studio\vc98\include\gl\gl.h(1152) : error C2085: 'APIENTRY' : not in formal parameter list	Didn't #include windows.h or included it after gl.h. Source files that use neither GLUT nor MFC, but which make calls to OpenGL, should: #include <windows.h> #include <gl gl.h=""></gl></windows.h>
d:c++\file.c(231) : warning C4305: 'initializing' : truncation from 'const double ' to 'float '	Floating-point constants (e.g., 1.0) default to type double. This is a harmless warning that can be disabled in Visual C++ with: #ifdef WIN32 #pragma warning(disable : 4305) #endif at the top of the source file.
file.obj : error LNK2001: unresolved external symbol impglMatrixMode@4 file.obj : error LNK2001: unresolved external symbol impglViewport@16 file.obj : error LNK2001: unresolved external symbol impglLoadIdentity@0	Didn't link with opengl32.lib, glu32.lib, or glut32.lib. <u>Section 2.060 above</u> describes how to inform the Visual C++ 6 linker about the location of the .lib files.

The dynamic link library OPENGL.dll could not be found in the specified path	Failure to correctly install .dll files. See <u>section 2.060 above</u> for information on where these files should be installed for your Windows system.
Nothing renders, just a blank window.	Mixed linkage against .lib files from both Microsoft and SGI can cause this. Make sure you specify either glut32.lib, glu32.lib opengl32.lib or glut.lib, glu.lib, and opengl.lib to the linker, but not a combination of the files from these two file sets.
LIBCD.lib(wincrt0.obj) : error LNK2001: unresolved external symbol _WinMain@16 Debug/test.exe : fatal error LNK1120: 1 unresolved externals Error executing link.exe.	Not an OpenGL question per se, but definitely a FAQ on comp.graphics.api.opengl due to the way GLUT works in Microsoft Windows. You should instruct your compiler to build a console application. It's trying to find the Win32 entry point, but your code wasn't written as a Win32 application.
Multiple access violations appear when running a Microsoft OpenGL MFC-based application.	Set the CS_OWNDC style in the PreCreate*() routines in the view class.
Floating–point exceptions occur at runtime. The application was built with Borland C.	Add the following to your app before you call any OpenGL functions: _control87(MCW_EM, MCW_EM); This is from Borland's own FAQ article #17197.

2.080 How do I initialize my windows, create contexts, etc.?

It depends on your windowing system. Here's some basic info, but for more details, refer to the documentation for your specific windowing system or a newsgroup devoted to programming in it.

GLUT

The basic code for creating an RGB window with a depth buffer, and an OpenGL rendering context, is as follows:

```
#include <GL/glut.h>
int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(0,0);
    glutCreateWindow("Simple");
    /* ... */
}
```

The calls to set the window size and position are optional, and GLUT uses a default size and location if they are left out.

X Windows

You can create an RGB window with a depth buffer in X Windows using the following code (taken from the *OpenGL Reference Manual*):

```
#include <GL/glx.h>
#include <GL/gl.h>
static Bool WaitForNotify(Display *d, XEvent *e, char *arg)
{
    return (e->type == MapNotify) && (e->xmap.window == (Window) arg);
}
static int sAttribList[] = {
    GLX_RGBA,
   GLX_RED_SIZE, 1,
   GLX_GREEN_SIZE, 1,
   GLX_BLUE_SIZE, 1,
   None };
int main(void)
{
   Display *dpy;
   XVisualInfo *vi;
   XSetWindowAttributes swa;
    Window win;
    GLXContext cx;
    XEvent event;
    int swap_flag = GL_FALSE;
    dpy = XOpenDisplay(0);
    if ((vi = glXChooseVisual(dpy, DefaultScreen(dpy), sAttribList)) == NULL) {
        fprintf(stderr, "ERROR: Can't find suitable visual!\n");
        return 0;
    }
    cx = glXCreateContext(dpy, vi, 0, GL_TRUE);
    swa.colormap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
        vi->visual, AllocNone);
    swa.border_pixel = 0;
    swa.event_mask = StructureNotifyMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 100, 100, 0,
        vi->depth, InputOutput,
        vi->visual,
        CWBorderPixel | CWColormap | CWEventMask,
        &swa);
    XMapWindow(dpy, win);
    XIfEvent(dpy, &event, WaitForNotify, (char *)win);
    glXMakeCurrent(dpy, win, cx);
    /* ... */
```

}

Microsoft Windows 9x/NT/2000

The window must be created with the following bits OR'd into the window style: WS_CLIPCHILDREN | WS_CLIPSIBLINGS. Do this either when CreateWindow is called (in a typical Win32 app) or during the PreCreateWindow function (in an MFC app).

Once the window is created (when a WM_CREATE message arrives or in the OnInitialUpdate callback), use the following code to set the pixel format, create a rendering context, and make it current to the DC.

```
// Assume:
// HWND hWnd;
HDC hDC = GetDC (hWnd);
PIXELFORMATDESCRIPTOR pfd;
memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion = 1;
pfd.dwFlags = PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW;
pfd.iPixelType = PFD_TYPE_RGBA;
pfd.cColorBits = 24;
pfd.cDepthBits = 32;
pfd.iLayerType = PFD_MAIN_PLANE;
int pixelFormat = ChoosePixelFormat(hDC, &pfd);
if (pixelFormat == 0) {
    // Handle error here
}
BOOL err = SetPixelFormat (hDC, pixelFormat, &pfd);
if (!err) {
    // Handle error here
}
hRC = wglCreateContext(hDC);
if (!hRC) {
    // Handle error here
}
err = wglMakeCurrent (hDC, hRC);
if (!err) {
    // Handle error here
}
```

You can then make the rendering context noncurrent, and release the DC with the following calls:

```
WglMakeCurrent(NULL,NULL);
ReleaseDC (hWnd, hDC);
```

2.090 How do I create a full-screen window?

Prior to GLUT 3.7, you can generate a full–screen window using a call to glutFullScreen(void). With GLUT 3.7 and later, a more flexible interface was added.

With glutGameModeString(), an application can specify a desired full-screen width and height, as well as the pixel depth and refresh rate. You specify it with an ASCII character string of the form [width]x[height]:[depth]@[hertz]. An application can use this mode if it's available with a call to glutEnterGameMode(void). Here's an example:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
glutGameModeString("640x480:16@60");
glutEnterGameMode();
```

Also, see the "Full Screen Rendering" section in the OpenGL game developer's FAQ.

2.100 What is the general form of an OpenGL program?

There are no hard and fast rules. The following pseudocode is generally recognized as good OpenGL form.

```
program_entrypoint
    // Determine which depth or pixel format should be used.
    // Create a window with the desired format.
    // Create a rendering context and make it current with the window.
    // Set up initial OpenGL state.
    // Set up callback routines for window resize and window refresh.
}
handle_resize
{
   glViewport(...);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
    // Set projection transform with glOrtho, glFrustum, gluOrtho2D, gluPerspective, etc
}
handle_refresh
{
    glClear(...);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // Set view transform with gluLookAt or equivalent
    // For each object (i) in the scene that needs to be rendered:
        // Push relevant stacks, e.g., glPushMatrix, glPushAttrib.
        // Set OpenGL state specific to object (i).
        // Set model transform for object (i) using glTranslatef, glScalef, glRotatef, a
        // Issue rendering commands for object (i).
        // Pop relevant stacks, (e.g., glPopMatrix, glPopAttrib.)
    // End for loop.
    // Swap buffers.
}
```

2.110 My window is blank. What should I do?

A number of factors can cause a blank window when you're expecting a rendering. A blank window is generally caused by insufficient knowledge of 3D graphics fundamentals, insufficient knowledge of basic OpenGL mechanisms, or simply a mistake in the code.

There are a number of **OpenGL** books and online resources as well.

What follows is a list some of the more common causes of the dreaded "Black Window Syndrome" and what to do to fix it.

• Your application may have made an erroneous call to OpenGL. Make liberal calls to glGetError(). You might create a macro or inline function, which does the following:

```
{
   GLint err = glGetError();
   if (err != GL_NO_ERROR) DisplayErrorMessage();
}
```

Place this code block after suspect groups of OpenGL function calls, and take advantage of the preprocessor, which will ensure that the calls can be eliminated easily in a production compile (i.e., #ifdef DEBUG...#endif).

glGetError() is the only way to tell whether you've issued an erroneous function call at runtime. If an OpenGL function generates an error, OpenGL won't process the offending function. This is often the cause of incorrect renderings or blank windows.

◆ Incorrect placement of *zFar* and *zNear* clipping planes with respect to the geometry can cause a blank window. The geometry is clipped and nothing is rendered. *zFar* and *zNear* clipping planes are parameters to the glOrtho(), gluOrtho2D(), glFrustum(), and gluPerspective() calls. For glFrustum() and gluPerspective(), it's important to remember that the *zNear* and *zFar* clipping planes are specified as distances in front of the eye. So, for example, if your eye is at (0,0,0), which it is in OpenGL eye coordinate space, and the *zNear* clipping plane is at 2.0 and all of your geometry is in a unit cube centered at the origin, the *zNear* plane will clip all of it and render nothing. You'll need to specify a ModelView transform to push your geometry back, such as a call to glTranslatef(0,0,−3).

Similarly, the *zFar* clipping plane might be a problem if it is placed at, for example, 10.0, and all of your geometry is further than 10.0 units from the eye.

• Incorrect transforms in general can cause a blank window. Your code is attempting to set the view and modeling transform correctly, but due to some problem, the net transformation is incorrect, and the geometry doesn't fall within the view volume. This is usually caused by a bug in the code or a lack of understanding of how OpenGL transforms work.

It's usually best to start simple and work your way to more complex transformations. Make code changes slowly, checking as you go, so you'll see where your mistakes came from.

♦ Another cause of the blank window is a failure to call glEnd() or failure to call glBegin(). Geometry that you specify with one of the glVertex*() routines must be wrapped with a glBegin()/glEnd() pair to be processed by OpenGL. If you leave out both glBegin() and glEnd(), you won't get an error, but nothing will render.

If you call glBegin(), but fail to call glEnd() after your geometry, you're not guaranteed that anything will render. However, you should start to see OpenGL errors once you call functions (e.g., glFlush()) that can't be called within a glBegin()/glEnd() pair. If you call

glEnd() but fail to call glBegin(), the glEnd() call will generate an error. Checking for errors is always a good idea.

- ♦ Failure to swap buffers in a double-buffered window can cause blank windows. Your primitives are drawn into the back buffer, but the window on the screen is blank. You need to swap buffers at the end of each frame with a call to SwapBuffers, glXSwapBuffers, or glutSwapBuffers.
- Failure to glClear() the buffers, in particular the depth buffer, is yet another cause. Call glClear() at the start of every frame to remedy this failue.
- Some OpenGL implementations have bugs that can cause blank windows or other incorrect rendering. Try your application on another implementation. Correct behavior on one or more other implementations is strong evidence of a bug in the first implementation.

2.120 The first frame is rendered correctly, but subsequent frames are incorrect or further away or I just get a blank screen. What's going on?

This is often caused by a failure to realize that OpenGL matrix commands multiply, rather than load over the top of the current matrix.

Most OpenGL programs start rendering a frame by setting the ModelView matrix to the identity with a call to glLoadIdentity(). The view transform is then multiplied against the identity matrix with, for example, a call to gluLookAt(). Many new programmers assume the gluLookAt() call will load itself onto the current matrix and therefore fail to initialize the matrix with the glLoadIdentity() call. Rendering successive frames in this manner causes successive camera transforms to multiply onto each other, which normally results in an incorrect rendering.

2.130 What is the AUX library?

Very important: Don't use AUX. Use GLUT instead.

The AUX library was developed by SGI early in OpenGL's life to ease creation of small OpenGL demonstration programs. It's currently neither supported nor maintained. Developing OpenGL programs using AUX is strongly discouraged. Use the GLUT instead. It's more flexible and powerful and is available on a wide range of platforms.

For related information, see the GLUT Section and SGI's GLUT FAQ.

2.140 What support for OpenGL does {Open,Net,Free}BSD or Linux provide?

The X Windows implementation, XFree86 4.0, includes support for OpenGL using Mesa or the OpenGL Sample Implementation. XFree86 is released under the XFree86 license. http://www.xfree86.org/

SGI has released the OpenGL Sample Implementation as open source. It can be built as an X server GLX implementation. It has been released under SGI Free Software License B. http://oss.sgi.com/projects/ogl-sample/

The Mesa 3D Graphics Library is an OpenGL clone that runs on many platforms, including

MS–DOS, Win32, *BSD and Linux. On PC UNIX platforms Mesa can be built to use GGI, X Windows, and as an X server GLX implementation. Mesa is hardware accelerated for a number of 3D graphics accelerators. Mesa 3.1 and later was released under an XFree86–style license. Versions prior to 3.1 were released under GPL. http://mesa3d.sourceforge.net/_

Utah–GLX is a hardware accelerated GLX implementation for the Matrox MGA–G200 and G–400, ATI 3D RAGE PRO, Intel i810, NVIDIA RIVA, and S3 ViRGE. Utah–GLX is based on Mesa. It is not clear what license Utah–GLX is released under. http://utah–glx.sourceforge.net/

Metro Link OpenGL and Extreme 3D are GLX extensions for Metro Link X servers. Metro Link OpenGL is a software implementation that can use accelerated X operations to gain a performance advantage over other software implementations. Metro Link Extreme 3D is a hardware–accelerated implementation for REALimage, GLINT GMX 1000, 2000, GLINT DMX, GLINT MX, GLINT TX, and Permedia 2 and 3. <u>http://www.metrolink.com/</u>

Xi Graphics 3D Accelerated–X is an X server with GLX support. Supported devices include: ATI Xpert 2000, ATI Rage Fury Pro, ATI Rage Fury, ATI Rage Magnum, ATI All–in–Wonder 128 (all ATI RAGE 128 I believe), 3Dlabs Oxygen VX1, 3Dlabs Permedia 3 Create! (Permedia 3), Diamond Stealth III S540, Diamond Stealth III S540 Extreme, Creative Labs 3D Blaster Savage4 (S3 Savage4), Number Nine SR9, 3Dfx Voodoo 3000, 3Dfx Voodoo 3500 software.

2.150 Where is OpenGL 1.2?

When this was written (early 2000), few OpenGL 1.2 implementations are available. Sun and IBM are shipping OpenGL 1.2. The OpenGL–like <u>Mesa</u> library also supports 1.2. The <u>OpenGL Sample Implementation</u> is also available.

Microsoft hasn't released OpenGL 1.2 yet. As of their most recent official announcement, it is to be included in a later Windows 2000 service pack. Once Microsoft releases OpenGL 1.2, you'll probably need a new driver to take advantage of its features.

Many OpenGL vendors running on Microsoft already support OpenGL 1.2 functionality through extensions to OpenGL 1.1.

OpenGL vendors that run on OS other than Microsoft will release OpenGL 1.2 on their own schedules.

The OpenGL 1.2 specification is available from <u>http://www.opengl.org</u>. The <u>red and blue</u> <u>books</u> have recently been revised to cover OpenGL 1.2 functionality.

2.160 What are the OpenGL Conformance Tests?

The OpenGL Conformance Tests are a suite of tests that the OpenGL ARB uses to certify an OpenGL implementation conforms to the OpenGL spec, and, after paying the licensing fee, is therefore entitled to call itself "OpenGL". The source code for the conformance tests can be licensed from the OpenGL ARB.

The conformance tests were recently upgraded to test the full OpenGL 1.2 functionality. They do not exercise extension entry points. They will, however, report the full list of extensions that an implementation claims to support.

covogl is a special conformance test that simply calls every standard entry point. It is a "coverage" test, meant to ensure that all entry points exist and don't crash. All the other tests are intended to test spec conformance for a specific rendering task.

The test mustpass.c tests a defined core of functionality that all OpenGL implementations must support. (You must be able to render a line," etc.) Vendors that fail other tests are still allowed to use the name "OpenGL", but they must be able to show that they understand the bugs, and are working to resolve the issue in a future release.

The ability to push and pop state is thoroughly tested. Each test that runs is of the form:

push state change state run test pop state check all state values (via glGet*()) to make sure they have returned to the default values.

Some tests have some built–in error that allows for some variation from the OpenGL specification. For example, OpenGL spec states that when rasterizing a triangle, the center of each rendered pixel must be within the mathematical boundary of the triangle. However, the conformance test for rasterizing triangles allows pixels to be as much as 1/2 pixel outside this boundary without reporting an error.

Conversely, some tests appear to test for more than the spec calls for. For example, the test for alpha test requires 10 percent (between 4 and 5 bits) precision to pass, whereas the spec calls for only a single bit of precision.

Some tests don't make sense if you are not intimately familiar with the spec. For example, the spec says it's perfectly OK to not antialias polygons when the user has requested it, and the conformance tests allow this. Another example is dithering; the spec allows for a great deal of implementation variety, including no dithering at all, and as a consequence, the conformance tests won't display an error if your implementation doesn't dither.

All tests support path levels that execute the same tests with a variety of state settings that should still produce the same result. For example, rendering a triangle with polygon stipple disabled should produce the same result as rendering it with polygon stipple enabled and a stipple pattern of all 1 bits. Again, this should be identical to rendering with blending enabled and a blend function of (GL_ONE,GL_ZERO). A number of path levels are available, each testing more and more complex combinations of state settings.

All tests are run on all available pixel formats or visual types, including (if available) color index.

All tests verify correct rendering with glReadPixels(). Some tests read the entire test window, while other read only a few key pixels. In general, the tests use GL_RGBA and GL_FLOAT as the *type and format*. However, the readpix.c test thoroughly tests all *type and format* combinations. If glReadPixels() is broken, all tests could fail.

If glReadPixels() is slow, the conformance tests can take a long time to run. Furthermore, since all tests run at all path levels on all available pixel formats and visuals, it could take several days of serial compute time to

run the entire test suite.

The conformance tests find many bugs. However, they don't guarantee a bug–free implementation. An implementation that passes the full suite of conformance tests might still be so buggy that many applications won't be able to run.

3 GLUT

3.010 What is GLUT? How is it different from OpenGL?

Because OpenGL doesn't provide routines for interfacing with a windowing system or input devices, an application must use a variety of other platform–specific routines for this purpose. The result is nonportable code.

Furthermore, these platform–specific routines tend to be full–featured, which complicates construction of small programs and simple demos.

GLUT is a library that addresses these issues by providing a platform–independent interface to window management, menus, and input devices in a simple and elegant manner. Using GLUT comes at the price of some flexibility.

A large amount of information on GLUT is at the GLUT FAQ: <u>http://reality.sgi.com/mjk/glut3/glut_faq.html</u>.

3.020 Should I use GLUT?

Your application might need to do things that GLUT doesn't allow, or it may need to use platform–specific libraries to accomplish nongraphical tasks. In this case, consider not using GLUT for your application's windowing and input needs, and instead use platform–specific libraries .

Ask yourself the following questions:

- Will my application run only on one platform?
- Do I need to use more than one rendering context?
- Do I need to share display lists or texture objects between rendering contexts?
- Do I need to use input devices that GLUT doesn't provide an interface for?
- Do I need to use platform-specific libraries for other tasks, such as sound or text?

If you answered yes to any of these questions, you need to evaluate whether GLUT is the right choice for your application.

3.030 I need to set up different tasks for left and right mouse button motion. However, I can only set one glutMotionFunc() callback, which doesn't pass the button as a parameter.

You can easily set up different tasks depending on the state of the SHIFT, ALT, and CTRL keys by checking their state with glutGetModifiers().

To set up different tasks for the left and right mouse buttons, you need to swap the motion function depending on which mouse button is in use. You can do this with a mouse function callback that you set with glutMouseFunc(). The first parameter to this routine will indicate which button caused the event (GLUT_LEFT, GLUT_MIDDLE, or GLUT_RIGHT). The second parameter indicates the button state (GLUT_UP or GLUT_DOWN).

To illustrate, here's an example glutMouseFunc() callback routine:

```
/* Declarations for our motion functions */
static void leftMotion (int x, int y);
static void rightMotion (int x, int y);
static void mouseCallback (int mouse, int state, int x, int y)
{
    if (state==GLUT_DOWN) {
        /* A button is being pressed. Set the correct motion function */
        if (button==GLUT_LEFT)
            glutMotionFunc (leftMotion);
        else if (button==GLUT_RIGHT)
            glutMotionFunc (rightButton);
    }
}
```

3.040 How does GLUT do...?

It is often desirable to find out how glut creates windows, handles input devices, displays menus, or any of a number of other tasks. The best way to find out how GLUT does something is to <u>download the GLUT source</u> and see how it is written.

3.050 How can I perform animations with GLUT?

GLUT allows your application to specify a callback routine for rendering a frame. You can force executing this routine by calling glutPostRedisplay() from another callback routine, and returning control to glutMainLoop().

To create an animation that runs as fast as possible, you need to set an idle callback with glutIdleFunc(). The callback you pass as a parameter will be executed by glutMainLoop() whenever nothing else is happening. From this callback, you call glutPostRedisplay().

To create a timed animation, use glutTimerFunc() instead of glutIdleFunc(). glutTimerFunc() will call your callback only after the specified time elapses. This callback disables itself, so for continuous updates, your callback must call both glutPostRedisplay(), then glutTimerFunc() again to reset the timer.

3.060 Is it possible to change a window's size *after* it's opened (i.e., after i called glutInitWindowSize(); and glutCreateWindow();)?

Once your code enters the glutMainLoop() and one of your callback routines is called, you can call glutReshapeWindow(int width, int height).

Note that glutReshapeWindow() doesn't instantly resize your window. It merely sends a message to GLUT to resize the window. This message is processed once you return to glutMainLoop().

3.070 I have a GLUT program that allocates memory at startup. How do I deallocate this memory when the program exits?

If the user exits your program through some input that you can catch, such as a key press or menu selection, the answer is trivial. Simply free the resources in the appropriate input event handler.

Usually, this question comes up because the user has killed the program through window frame controls, such as the Microsoft Windows Close Window icon in the upper right corner of the title bar. In this case, your program won't get a GLUT event indicating the program is exiting. In fact, when the window is destroyed, glutMainLoop() simply calls exit(0).

For simple resources such as memory deallocation, this should not be a problem. The OS will free any memory that the process was using.

Of greater concern is prompting the user to save work or flushing data held in software buffers to files.

When using C++, the simplest solution to this problem is to wrap your GLUT application inside of a C++ class and create it with global scope. The C++ language guarantees that the class' destructor is called when the object goes out of scope.

Another option is to use the ANSI C/C++ atexit() call to specify the address of a function to execute when the program exits. You need to declare your buffers and data pointers with global scope so they're accessible to the atexit() callback routine. More information can be found in any ANSI C/C++ reference. atexit() is only available with C/C++.

One final option is to <u>hack the GLUT source</u>, and add an explicit callback to your code when glutMainLoop() catches the destroy window event/message. This is distasteful, for it means you must now include the entire hacked glutMainLoop() function in your application.

3.080 How can I make my GLUT program detect that the user has closed the window?

The same way as the previous section 3.070 shows.

3.090 How can I make glutMainLoop() return to my calling program?

glutMainLoop() isn't designed to return to the calling routine. GLUT was designed around the idea of an event–driven application, with the exit method being captured through an input event callback routine, such as a GLUT menu or keyboard callback handler.

If you insist on returning to your program from glutMainLoop(), there is only one way to do so. You need to download the GLUT source and hack gluMainLoop() to do what you want it to. Then compile and link into your program this hacked version of glutMainLoop(). <u>Steve</u> Baker has a Web site with the details on how to hack glutMainLoop() to eliminate this problem.

3.100 How do I get rid of the console window in a Windows GLUT application?

With Visual C++ 6.0, go to the Project menu, Settings... dialog. Select the Link tab. In the Project options edit box, add /SUBSYSTEM:WINDOWS /ENTRY:mainCRTStartup to the end of the present text. Link options are similar for other Windows compilers.

3.110 My GLUT question isn't answered here. Where can I get more info?

SGI's GLUT FAQ is an excellent source of information on GLUT.

4 GLU

4.010 What is GLU? How is it different from OpenGL?

If you think of OpenGL as a low-level 3D graphics library, think of GLU as adding some higher-level functionality not provided by OpenGL. Some of GLU's features include:

- Scaling of 2D images and creation of mipmap pyramids
- Transformation of object coordinates into device coordinates and vice versa
- Support for NURBS surfaces
- Support for tessellation of concave or bow tie polygonal primitives
- Specialty transformation matrices for creating perspective and orthographic projections, positioning a camera, and selection/picking
- Rendering of disk, cylinder, and sphere primitives
- ◆ Interpreting OpenGL error values as ASCII text

The best source of information on GLU is the OpenGL <u>red and blue books</u> and the GLU specification, which you can obtain from <u>the OpenGL org Web page</u>.

4.020 How does GLU render sphere, cylinder, and disk primitives?

There is nothing special about how GLU generates these primitives. You can easily write routines that do what GLU does. You can also download <u>the Mesa source</u>, which contains a GLU distribution, and see what these routines are doing.

The GLU routines approximate the specified primitive using normal OpenGL primitives, such as quad strips and triangle fans. The surface is approximated according to user parameters. The vertices are generated using calls to the sinf() and cosf() math library functions.

If you are interested in rendering cylinders and tubes, you'll want to examine <u>the GLE</u> <u>library</u>. GLE comes as part of the GLUT distribution.

4.030 How does gluPickMatrix() work?

It simply translates and scales so that the specified pick region fills the viewport. When specified on the projection matrix stack, prior to multiplying on a normal projection matrix (such as gluPerspective(), glFrustum(), glOrtho(), or gluOrtho2D()), the result is that the view volume is constrained to the pick region. This way only primitives that intersect the pick region will fall into the view volume. When glRenderMode() is set to GL_SELECT, these primitives will be returned.

4.040 How do I use GLU tessellation routines?

GLU provides tessellation routines to let you render concave polygons, self-intersecting polygons, and polygons with holes. The tessellation routines break these complex primitives up into (possibly groups of) simpler, convex primitives that can be rendered by the OpenGL API. This is done by providing the data of the simpler primitives to your application from callback routines that your application must provide. Your app can then send the data to OpenGL using normal API calls.

An example program is available in the GLUT distribution under progs/redbook/tess.c. (<u>Download the GLUT distribution</u>).

The usual steps for using tessellation routines are:

1. Allocate a new GLU tessellation object:

GLUtesselator *tess = gluNewTess();

2. Assign callbacks for use with this tessellation object:

gluTessCallback (tess, GLU_TESS_BEGIN, tcbBegin); gluTessCallback (tess, GLU_TESS_VERTEX, tcbVertex); gluTessCallback (tess, GLU_TESS_END, tcbEnd);

2a. If your primitive is self-intersecting, you must also specify a callback to create new vertices:

gluTessCallback (tess, GLU_TESS_COMBINE, tcbCombine);

3. Send the complex primitive data to GLU:

```
// Assumes:
// GLdouble data[numVerts][3];
// ...and assumes the array has been filled with 3D vertex data.
gluTessBeginPolygon (tess, NULL);
gluTessBeginContour (tess);
for (i=0; i<sizeof(data)/(sizeof(GLdouble)*3);i++)
gluTessVertex (tess, data[i], data[i]);
gluTessEndContour (tess);
gluEndPolygon (tess);
```

4. In your callback routines, make the appropriate OpenGL calls:

```
void tcbBegin (GLenum prim);
{
   glBegin (prim);
}
void tcbVertex (void *data)
{
  glVertex3dv ((GLdouble *)data);
}
void tcbEnd ();
{
  glEnd ();
}
void tcbCombine (GLdouble c[3], void *d[4], GLfloat w[4], void **out)
{
   GLdouble *nv = (GLdouble *) malloc(sizeof(GLdouble)*3);
  nv[0] = c[0];
  nv[1] = c[1];
  nv[2] = c[2];
   *out = nv;
}
```

The above list of steps and code segments is a bare–bones example and is not intended to demonstrate the full capabilities of the tessellation routines. By providing application–specific data as parameters to gluTessBeginPolygon() and gluTessVertex() and handling the data in the appropriate callback routines, your application can color and texture map these primitives as it would a normal OpenGL primitive.

4.050 Why aren't my tessellation callback routines called?

Normally your tessellation callback routines are executed when you call gluEndPolygon(). If they are not being called, an error has occurred. Typically this is caused when you haven't defined a GLU_TESS_COMBINE* callback for a self–intersecting primitive.

You might try defining a callback for GLU_TESS_ERROR to see if it's called.

4.060 How do I use GLU NURBS routines?

The GLU NURBS interface converts the B–Spline basis control points into Bezier basis equivalents and calls directly to the OpenGL Evaluator routines to render the surface.

An example program is available in the GLUT distribution under progs/redbook/surface.c. (Download the GLUT distribution).

4.070 How do I use gluProject() and gluUnProject()?

Both routines take a ModelView matrix, Projection matrix, and OpenGL Viewport as parameters.

gluProject() also takes an XYZ–object space coordinate. It returns the transformed XYZ window (or device) coordinate equivalent.

gluUnProject() does the opposite. It takes an XYZ window coordinate and returns the back-transformed XYZ object coordinate equivalent.

The concept of window space Z is often confusing. It's the depth buffer value expressed as a GL double in the range 0.0 to 1.0. Assuming a default glDepthRange(), a window coordinate with a Z value of 0.0 corresponds to an eye coordinate located on the *zNear* clipping plane. Similarly, a window space Z value of 1.0 corresponds to an eye space coordinate located on the *zFar* plane. You can obtain any window space Z value by reading the depth buffer with glReadPixels().

5 Microsoft Windows Specifics

5.010 What's a good source for Win32 OpenGL programming information?

Samuel Paik has created a large repository of <u>links to OpenGL information on Microsoft</u> <u>Web sites</u>.

See Blaine Hodge's web page. Be aware that some examples on this page use the AUX library, which is not recommended.

5.020 I'm looking for a Wintel OpenGL card in a specific price range, any suggestions?

The consumer-level 3D graphics marketplace moves fast. Any information placed in this FAQ would be soon outdated.

You might post a query on this topic to the comp.graphics.api.opengl newsgroup, or one of the many newsgroups devoted to Wintel-based 3D games. You might also do a Web search.

Tom's Hardware Guide and Fast Graphics have a lot of information on current graphics cards.

5.030 How do I enable and disable hardware rendering on a Wintel card?

Currently, OpenGL doesn't contain a switch to enable or disable hardware acceleration. Some vendors might provide this capability with an environment variable or software switch.

If you install your graphics card, but don't see hardware accelerated rendering check for the following:

- Did you install the device driver / OpenGL Installable Client Driver (ICD)? (<u>How do</u> <u>I do that</u>?)
- ◆ Is your desktop in a supported color depth? (Usually 16– and 32–bit color are accelerated. See your device vendor for details.)
- Did your application select an accelerated pixel format?

You might also have acceleration problems if you're trying to set up a multimonitor configuration. Hardware accelerated rendering might not be supported on all (or any) devices in this configuration.

To force software rendering from your application, choose a pixel format that is not hardware accelerated. To do this, you can not use ChoosePixelFormat(), which always selects a hardware accelerated pixel format when one is available. Instead, use DescribePixelFormat() to iterate through the list of available pixel formats. Any format with the PFD_GENERIC_FORMAT attribute bit set will not be hardware accelerated.

An example of iterating over available pixel formats can be found here.

A less tasteful method to disable hardware acceleration is to move or rename your OpenGL ICD.

Also, check your device's documentation to see if your device driver supports disabling

hardware acceleration by a dialog box.

5.040 How do I know my program is using hardware acceleration on a Wintel card?

OpenGL doesn't provide a direct query to determine hardware acceleration usage. However, this can usually be inferred by using indirect methods.

If you are using the Win32 interface (as opposed to GLUT), call DescribePixelFormat() and check the returned dwFlags bitfield. If PFD_GENERIC_ACCELERATED is clear and PFD_GENERIC_FORMAT is set, then the pixel format is only supported by the generic implementation. Hardware acceleration is not possible for this format. For hardware acceleration, you need to choose a different format.

If glGetString(GL_VENDOR) returns something other than "Microsoft Corporation", it means you're using the board's ICD. If it returns "Microsoft Corporation", this implies you chose a pixel format that your device can't accelerate. However, glGetString(GL_VENDOR) also returns this if your device has an MCD instead of an ICD, which means you might still be hardware accelerated in this case.

Another way to check for hardware acceleration is to temporarily remove or rename the ICD, so it can't be loaded. If performance drops, it means you were hardware accelerated before. Don't forget to restore the ICD to its original location or name. (To find your ICD file name, run the regedit utility and search for a key named "OpenGLdrivers".)

You can also gather performance data by rendering into the back buffer and comparing the results against known performance statistics for your device. This method is particularly useful for devices that revert to software rendering for some state combinations or OpenGL features. See the section on performance for more information.

5.050 Where can I get the OpenGL ICD for a Wintel card?

If your device supports OpenGL, the manufacturer should provide an ICD (commonly referred to as the device driver) for it. After you install the ICD, your OpenGL application can use the device's hardware capabilities.

If your device didn't come with an ICD on disk, you'll need to check the manufacturer's Web page to see where you can download the latest drivers. The chip manufacturer will probably have a more current ICD than the board manufacturer. Find the device driver download page, get the latest package for your device, and install it per the instructions provided.

Check <u>Reactor Critical</u> for nVidia device drivers. They often have more current and better performing OpenGL device drivers than nVidia makes available from their web page.

GLsetup, a free utility, is available. According to the GLsetup Web page, it "detects a user's 3D graphics hardware and installs the correct device drivers." Windows 2000 device drivers might not be supported. You can get it from <u>http://www.glsetup.com</u>.

5.060 I'm using a Wintel card, and an OpenGL feature doesn't seem to work. What's going on?

It could simply be a bug in your code. However, if the same code works fine on another OpenGL implementation, this implies the problem is in your graphics device or its ICD. See

the previous question for information on obtaining the latest ICD for your device.

5.070 Can I use OpenGL with DirectDraw?

Moxing OpenGL rendering calls with rendering calls from other APIs (such as DirectDraw) in the same window won't work on some drivers, and is therefore unportable. I don't recommended it.

5.080 Can I use use DirectDraw to change the screen resolution or desktop pixel depth?

You can create a window and use DirectDraw to change the display resolution and/or pixel depth. Then, get the window's DC and create an OpenGL context from it. This is known to work on some devices.

While we're on the subject, Microsoft doesn't require, and consequently does not test for, the ability to render OpenGL into a DirectDraw surface. Just because you can get a surface's DC does not mean that OpenGL rendering is supported. Always check for error returns when creating contexts or maining them current.

5.090 My card supports OpenGL, but I don't get acceleration regardless of which pixel format I use.

Are you in 8bpp? There are few 3D accelerators for PCs that support acceleration in 8bpp.

5.100 How do I get hardware acceleration?

The pixel format selects hardware acceleration. Pay attention to the flags GENERIC_FORMAT and GENERIC_ACCELERATED. You want both of them on if you're using a 3D–DDI or an MCD and neither on if you are using an ICD. You may have to iterate using DescribePixelFormat() instead of only using ChoosePixelFormat().

5.110 Why doesn't OpenGL hardware acceleration work with multiple monitors?

In Windows 98, Microsoft decided to disable OpenGL hardware acceleration when multiple monitors are enabled. In Windows NT 4.0, some drivers support multiple monitors when using identical (or nearly identical) cards. I don't believe multiple monitors and hardware accelerated OpenGL work with different types of cards. I don't know the story with Windows 2000, but it's likely to be similar to Windows NT 4.0.

5.120 Why does my MFC window flash, even though I'm using double buffering?

Your view class should have an OnEraseBkgnd() event handler. You can eliminate flashing by overriding this function and returning TRUE. This tells Windows that you've cleared the window. Of course, you didn't really clear the window. However, overriding the function keeps Microsoft from trying to do it for you, and should prevent flashing.

5.121 Why does my double buffered window appear incomplete or contain black stripes?

This is a problem with MS OpenGL. The bug is in the generic code, or possibly in MS Windows itself, because it occurs even with pure software rendering. To work around the bug, try one of these two methids:

- Create the OpenGL drawing window, but don't make it visible immediately. Get the screen size and set the window's size to be the same as the screen. Now set the pixel format and create the HGLRC. Set the window's size back to whatever it should be and make the window visible. This hack is invisible to the user, but doesn't always work.
- When the window is resized larger, destroy and re-create the window. This is really ugly and visible to the user, but it seems to always work.

5.130 What's the difference between opengl.dll and opengl32.dll?

According to OpenGL Reference Manual editor Dave Shreiner:

"Unless there's an absolutely compelling reason ... I really would suggest using opengl32.dll, and letting the old opengl.dll thing die.

"opengl.dll comes from the now totally unsupported OpenGL for Windows release of OpenGL for Microsoft Windows by SGI. We stopped supporting that release over two years — like no one ever touches the code. ...

"Now, why use opengl32.dll? For the most part, SGI provides Microsoft with the ICD kit, sample drivers, and software OpenGL implementation that you find there. Its really the same code base (with fixes and new features) as the opengl.dll, its only that we got Microsoft to ship and support it (in a manner of speaking)."

More information on linking with opengl.dll can be found here.

5.140 Should I use Direct3D or OpenGL?

A good comparison of the two can be found here.

5.150 What do I need to know to use OpenGL with MFC?

You need to be familiar with both OpenGL and the Microsoft Foundation Class (MFC). An online MFC reference is available, <u>the MFC FAQ</u>. You don't need to be an MFC guru to add OpenGL to an MFC application. Familiarity with C++ can make mastering MFC easier, and the more you know about MFC, the more you can concentrate on your OpenGL code. If you have only a rudimentary knowledge of MFC, look at the <u>downloadable source code example</u> <u>below</u>, and look at the steps necessary to recreate it.

Samuel Paik's repository of <u>links to OpenGL information on Microsoft Web sites</u> also has information on using OpenGL and MFC.

Here's a list of books that might be helpful.

OpenGL Programming for Windows 95 and Windows NT, by Ron Fosner. This is also known as the white book. It contains good information on using OpenGL in Microsoft Windows. Much of the information in it can be found on the MSDN Web site, but the book presents the information in a more logical and easily digestable format, and comes with good demos.

Opengl Superbible: The Complete Guide to Opengl Programming for Windows NT and Windows 95, by Richard S. Wright and Michael Sweet. This book contains a chapter on

OpenGL and MFC.

MFC Programming with Visual C++ 6 *Unleashed*, by David White, et al. The book contains a short chapter on OpenGL and focuses more on DirectX.

5.160 How can I use OpenGL with MFC?

To add OpenGL to an MFC application, you need to do at least the following:

- ♦ Add glu32.lib opengl32.lib to the list of object/library modules to link with.
- When your View class's OnInitialUpdate() function is called, set the pixel format and create a rendering context as you would for a Win32 application.
- Render your OpenGL scene when the View needs to be updated, or add a Run message handler to your Application class that updates when idle.

You can render OpenGL into any CWnd object, including frame windows, dialog boxes, and controls.

Download this example, which demonstrates OpenGL in a CStatic form control. This code uses a CGlView class that takes any CWnd as a parameter to its constructor. Rather than create a View derived from a CFormView, you could just as easily create an SDI application, and pass "this" (an instantiation of a CView) as a parameter to the constructor. Follow these steps to recreate this sample code using Microsoft Visual C++ v6.0:

- 1. If you haven't done so already, download the example. You'll need to borrow code from it in the steps that follow.
- 2. Create an MFC application using the AppWizard. Use defaults, except derive your View class from a CFormView. The project will open in the resource editor. Add a FORM control to the open CFormView. Call it IDC_OPENGLWIN.
- 3. Select Project->Settings...->Link, and add glu32.lib opengl32.lib to the list of objects/library modules.
- 4. Select Project->Add To Peoject->Files... and add the CGlView.cpp OpenGL view class source file from the above example code.
- 5. From the class view, right click your application's View class and select Add Member Variable... Set the variable type to CGlView *, the name to m_pclGLView, and the access to Private.
- 6. In your application's View class header file, add #include "CGlView.h" just before the class definition.
- 7. Find the global declaration of "theApp". Immediately after this declaration, add two new global variables:

```
CGlView *g_pclGLView = NULL;
MSG msg;
```

- In the wizard bar, set the application's View class, set the filter to All Class Members, and select the OnInitialUpdate member function.
- For the CGlView class to work, it needs a CWnd to initialize OpenGL for that window. For this example, our CWnd is the CStatic FORM control we added in step 1. After the existing code in this function, add the following:

```
CStatic pclStatic = (CStatic *)GetDlgItem(IDC_OPENGLWIN);
m_pclGLView = new CGlView(pclStatic);
```

• Open the class wizard with View->ClassWizard. From the message map tab, select your project's Application class. Add a function handler for the Run message. Replace the generated code with the Run message handler from the downloaded example.

5.170 Is OpenGL inherently slower when used with MFC?

Nothing in MFC guarantees a slow–running OpenGL application. However, some poorly written MFC applications might run slowly. This is a possibility in any development environment and is not specific to OpenGL. Here are some things to look out for:

- 1. Build the application as Release instead of Debug. Disable the TRACE debugging feature.
- 2. Avoid MFC classes such as CArray, CMap, and CList that perform inefficient data copies.
- 3. You may be able to improve performance by avoiding the WM_PAINT message. See the question above for example source that does this.
- 4. MFC classes are general purpose. For maximum performance, write a tuned implementation of an MFC class.
- 5. Use standard efficient programming techniques such as avoiding redundant calls, etc.

5.180 Where can I find MFC examples?

This FAQ contains an example.

Alan Oursland, Using OpenGL in Visual C++ Version 4.x, DevCentral Learning Center, <u>http://devcentral.iftech.com/learning/tutorials/mfc-win32/opengl/</u>. This is good but dated. It will get you started with a SDI MFC OpenGL application.

Mahesh Venkitachalam, OpenGL Code, <u>http://home.att.net/~bighesh/ogl.html</u>. Mahesh presents OpenGL in a no application wizard, minimal MFC program along with some OpenGL techniques.

Roman Podobedov, Skeleton of OpenGL program for Windows (MFC). <u>http://madli.ut.ee/~romka/opengl/demos/win32_eng.htm.</u> This is a minimal MFC program with no controls or application wizard.

Paul Martz, Generating Random Fractal Terrain.

<u>http://www.gameprogrammer.com/fractal.html</u>. This is a good example of the MFC SDI approach. However, the primary focus of the example is terrain, to which OpenGL and MFC take a back seat.

[5] Pierre Alliez, Starting OpenGL in a Dialog. http://codeguru.earthweb.com/opengl/texture_mapping.shtml.

Pierre Alliez, Starting Rendering Modes. <u>http://www.codeguru.com/opengl/start.shtml</u>. This is a splitter window example.

Pierre Alliez, How to snap an OpenGL client and send it to the clipboard, <u>http://codeguru.earthweb.com/opengl/snap.shtml</u>.

Pierre Alliez, A small VRML viewer using OpenGL and MFC.

OpenGL FAQ and Troubleshooting Guide

http://www.codeproject.com/opengl/wrl_viewer.asp.

Uwe Kotyczka, Enu.zip, <u>http://ws56.tinst.uni–jena.de/opengl_en.html</u>. This rather large and impressive MFC contribution demonstrates, multiple OpenGL views, rubber banding, color ramp, mouse trackball type control, OpenGL printing, etc., in a MFC MDI and SDI framework. This was built with VC++ 6.0 (SP4).

5.190 What do I need to know about mixing WGL and GDI calls?5.200 Why does my code crash under Windows NT or 2000 but run fine under 9x?5.210 How do I properly use WGL functions?

Charles E. Hardwidge has <u>tutorial articles and examples</u> for download that address these issues. The idea is to use WGL, GDI, and OpenGL functions such that the Microsoft OpenGL ICD mechanism isn't assumed.

6 Windows, Buffers, and Rendering Contexts

6.010 How do I use overlay planes?

Overlays planes allow layered rendering. They support a transparent color to let rendering in underlying planes show through. Any combination of overlay and underlay planes are possible, but a typical implementation consists of a single overlay plane along with the main framebuffer plane. It's common for overlay planes to be available only in color index mode, though RGB overlays are available on some devices. The transparent color is normally (0,0,0) for RGB overlays and index 0 for color index overlays. Rendering to the overlay plane is non-destructive to the main plane and vice versa.

How you access overlay planes depends on the windowing system interface. While GLUT 3.7 has entry points defined for the use of overlays, currently the entry points are disabled. To use overlays, your application needs to use WGL, GLX, or another platform–specific interface.

For both WGL and GLX, the basic idea is the same: Create two rendering contexts, one for the main plane and another for the overlay planes. Once they're created, make either context current, depending on whether your application needs to render to the overlay or the main plane.

In WGL, use ChoosePixelFormat() to select a pixel format with an overlay or underlay plane. When your application calls this function, use the bReserved field of the PIXELFORMATDESCRIPTOR to indicate the desired overlay or underlay plane. A value of 1 indicates one overlay plane. The iLayerType field needs to be set to PFD_MAIN_PLANE.

After setting the pixel format, create the rendering context for the main plane as usual, then create a second rendering context for the overlay plane as follows:

```
HGLRC hORC;
hORC = wglCreateLayerContext (hDC, a);
```

Check the return value the same way you would for a call to wglCreateContext(). A value of NULL indicates failure.

In GLX, use glXChooseVisual() to obtain a list of visuals with overlays. Add GLX_LEVEL to the attribute list, followed by the fullword indication of the desired level. Positive values indicate overlay; negative values indicate underlay. A value of 0 indicates the main plane, which is the default. A typical application will call glXChooseVisual() twice, once with GLX_LEVEL set to 0, and again with GLX_LEVEL set to 1. After each call, choose one of the returned visuals to create a rendering context.

When your application can't find a pixel format or visual that supports overlay, there are two common causes. Overlay might not be available on your platform, or you could be asking for an RGB overlay when only color index is available.

7 Interacting with the Window System, Operating System, and Input Devices

7.010 How do I obtain the window width and height or screen max width and height?

To obtain the window size on Win32, use the following code:

RECT rect; HWND hwnd; GetClientRect(hwnd, &rect); /* rect.top and rect.left will always be 0, 0, respectively. The width and height are in rect.right and rect.bottom. */

For the screen size in pixels on Win32:

int width = GetSystemMetrics(SM_CXSCREEN); int height = GetSystemMetrics(SM_CYSCREEN);

To obtaining the screen and window width and height using GLUT:

```
int screenWidth, screenHeight, windowWidth, windowHeight;
screenWidth = glutGet(GLUT_SCREEN_WIDTH);
screenHeight = glutGet(GLUT_SCREEN_HEIGHT);
windowWidth = glutGet(GLUT_WINDOW_WIDTH);
windowHeight = glutGet(GLUT_WINDOW_HEIGHT);
```

7.020 What user interface system should I use?

Most user interface (UI) systems, such as Motif, are restricted to a subset of operating systems that support OpenGL. GLUT is available on a variety of windowing systems, and it supports hierarchical menus. However, this fills the UI requirements of only simple applications.

The GLUI toolkit implements buttons, checkboxes, radio buttons, and spinners, which are layered on top of GLUT. Therefore, this UI is window system independent. Go to http://www.cs.unc.edu/~rademach/glui/ for more details.

7.030 How can I use multiple monitors?

Many OpenGL implementations support multiple monitor configurations. These come in a variety of different flavors:

- One display is hardware accelerated, the rest are not.
- All heads are accelerated as long as OpenGL windows do not span display boundaries.
- As above, with support for OpenGL windows that span multiple displays.
- All of the above, with support for stereo.
- All of the above, with support for heterogeneous graphics cards vendors.

Some Macintosh configurations, such as the ATI Rage 128 Mobility, allow dual displays. However, hardware acceleration is disabled if the OpenGL window spans both displays. If

OpenGL FAQ and Troubleshooting Guide

the window lies completely on one display or the other, full hardware acceleration is available.

Microsoft operating systems allow two OpenGL devices in a single system, but only the primary device is hardware accelerated. To work around this issue, many vendors have provided their own multiple monitor solutions, so that hardware acceleration is available on both displays.

3Dlabs supports multi-head on GVX1, GVX210 and GVX420. The latter two cards are a single AGP card with dual monitor output. The GVX1 supports one display per device, and comes in both AGP and PCI versions to support signle–AGP slot systems.

HP Visualize Center and Visualize Workgroup allow full hardware acceleration in windows spanning two or more displays under HP–UX. Visualize Center blends multiple projector displays seamlessly on a wall–sized screen. Stereo is supported to produce a completely immersive environment.

Matrox under Linux and Microsoft operating systems supports DualHead, digital flat panel, and TV out for the G400, DualHead for the G450, and multiple monitors for the G200 series. . In addition to supporting a single logical display, their Microsoft Windows device drivers also support a dual desktop mode.

The *NVIDIA* Quadro2 MXR and GeForce2 GTS devices both support two displays from a single card. Under Miscrosoft Windows operating systems, the Display Properties dialog features a TwinView tab that allows the user to configure the displays. In Vertical Span mode, a single logical desktop spans both monitors. In this configuration, an application that creates a window the size of the screen will automatically create a window that fills both monitors. No change to the application is required, and both displays are accelerated in hardware. NVIDIA Linux drivers do not currently support hardware accelerated rendering, according to the <u>NVIDIA Linux FAQ</u>.

All versions of *Sun* OpenGL for Solaris on SPARC support accelerated multihead configurations provided the display is OpenGL accelerated. With Solaris 8 and Sun OpenGL 1.2.1 an accelerated OpenGL window can span multiple heads provided the display devices are the same and the device is OpenGL accelerated. Sun OpenGL is available from: http://sun.com/software/graphics/opengl

8 Using Viewing and Camera Transforms, and gluLookAt()

8.010 How does the camera work in OpenGL?

As far as OpenGL is concerned, there is no camera. More specifically, the camera is always located at the eye space coordinate (0., 0., 0.). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation.

8.020 How can I move my eye, or camera, in my scene?

OpenGL doesn't provide an interface to do this using a camera model. However, the GLU library provides the gluLookAt() function, which takes an eye position, a position to look at, and an up vector, all in object space coordinates. This function computes the inverse camera transform according to its parameters and multiplies it onto the current matrix stack.

8.030 Where should my camera go, the ModelView or Projection matrix?

The GL_PROJECTION matrix should contain only the projection transformation calls it needs to transform eye space coordinates into clip coordinates.

The GL_MODELVIEW matrix, as its name implies, should contain modeling and viewing transformations, which transform object space coordinates into eye space coordinates. Remember to place the camera transformations on the GL_MODELVIEW matrix and never on the GL_PROJECTION matrix.

Think of the projection matrix as describing the attributes of your camera, such as field of view, focal length, fish eye lens, etc. Think of the ModelView matrix as where you stand with the camera and the direction you point it.

The game dev FAQ has good information on these two matrices.

Read Steve Baker's article on <u>projection abuse</u>. This article is highly recommended and well–written. It's helped several new OpenGL programmers.

8.040 How do I implement a zoom operation?

A simple method for zooming is to use a uniform scale on the ModelView matrix. However, this often results in clipping by the *zNear* and *zFar* clipping planes if the model is scaled too large.

A better method is to restrict the width and height of the view volume in the Projection matrix.

For example, your program might maintain a zoom factor based on user input, which is a floating-point number. When set to a value of 1.0, no zooming takes place. Larger values result in greater zooming or a more restricted field of view, while smaller values cause the opposite to occur. Code to create this effect might look like:

static float zoomFactor; /* Global, if you want. Modified by user input. Initially 1.0 * $\,$

```
/* A routine for setting the projection matrix. May be called from a resize
    event handler in a typical application. Takes integer width and height
    dimensions of the drawing area. Creates a projection matrix with correct
    aspect ratio and zoom factor. */
void setProjectionMatrix (int width, int height)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective (50.0*zoomFactor, (float)width/(float)height, zNear, zFar);
    /* ...Where 'zNear' and 'zFar' are up to you to fill in. */
}
```

Instead of gluPerspective(), your application might use glFrustum(). This gets tricky, because the *left*, *right*, *bottom*, and *top* parameters, along with the *zNear* plane distance, also affect the field of view. Assuming you desire to keep a constant *zNear* plane distance (a reasonable assumption), glFrustum() code might look like this:

```
glFrustum(left*zoomFactor, right*zoomFactor,
    bottom*zoomFactor, top*zoomFactor,
    zNear, zFar);
```

glOrtho() is similar.

8.050 Given the current ModelView matrix, how can I determine the object-space location of the camera?

The "camera" or viewpoint is at (0., 0., 0.) in eye space. When you turn this into a vector $[0\ 0\ 0\ 1]$ and multiply it by the inverse of the ModelView matrix, the resulting vector is the object–space location of the camera.

OpenGL doesn't let you inquire (through a glGet* routine) the inverse of the ModelView matrix. You'll need to compute the inverse with your own code.

8.060 How do I make the camera "orbit" around a point in my scene?

You can simulate an orbit by translating/rotating the scene/object and leaving your camera in the same place. For example, to orbit an object placed somewhere on the Y axis, while continuously looking at the origin, you might do this:

If you insist on physically orbiting the camera position, you'll need to transform the current camera position vector before using it in your viewing transformations.

In either event, I recommend you investigate gluLookAt() (if you aren't using this routine already).

8.070 How can I automatically calculate a view that displays my entire model? (I know the bounding sphere and up vector.)

8 Using Viewing and Camera Transforms, and gluLookAt()

The following is from a posting by Dave Shreiner on setting up a basic viewing system:

First, compute a bounding sphere for all objects in your scene. This should provide you with two bits of information: the center of the sphere (let (c.x, c.y, c.z) be that point) and its diameter (call it "diam").

Next, choose a value for the *zNear* clipping plane. General guidelines are to choose something larger than, but close to 1.0. So, let's say you set

```
zNear = 1.0;
zFar = zNear + diam;
```

Structure your matrix calls in this order (for an Orthographic projection):

```
GLdouble left = c.x - diam;
GLdouble right = c.x + diam;
GLdouble bottom c.y - diam;
GLdouble top = c.y + diam;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bottom, top, zNear, zFar);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

This approach should center your objects in the middle of the window and stretch them to fit (i.e., its assuming that you're using a window with aspect ratio = 1.0). If your window isn't square, compute *left, right, bottom,* and *top,* as above, and put in the following logic before the call to glOrtho():

```
GLdouble aspect = (GLdouble) windowWidth / windowHeight;
if ( aspect < 1.0 ) { // window taller than wide
    bottom /= aspect;
    top /= aspect;
} else {
    left *= aspect;
    right *= aspect;
}
```

The above code should position the objects in your scene appropriately. If you intend to manipulate (i.e. rotate, etc.), you need to add a viewing transform to it.

A typical viewing transform will go on the ModelView matrix and might look like this:

GluLookAt (0., 0., 2.*diam, c.x, c.y, c.z, 0.0, 1.0, 0.0);

8.080 Why doesn't gluLookAt work?

This is usually caused by incorrect transformations.

Assuming you are using gluPerspective() on the Projection matrix stack with *zNear* and *zFar* as the third and fourth parameters, you need to set gluLookAt on the ModelView matrix stack, and pass parameters so your geometry falls between *zNear* and *zFar*.

It's usually best to experiment with a simple piece of code when you're trying to understand viewing transformations. Let's say you are trying to look at a unit sphere centered on the origin. You'll want to set up your transformations as follows:

It's important to note how the Projection and ModelView transforms work together.

In this example, the Projection transform sets up a 50.0–degree field of view, with an aspect ratio of 1.0. The *zNear* clipping plane is 3.0 units in front of the eye, and the *zFar* clipping plane is 7.0 units in front of the eye. This leaves a Z volume distance of 4.0 units, ample room for a unit sphere.

The ModelView transform sets the eye position at (0.0, 0.0, 5.0), and the look-at point is the origin in the center of our unit sphere. Note that the eye position is 5.0 units away from the look at point. This is important, because a distance of 5.0 units in front of the eye is in the middle of the Z volume that the Projection transform defines. If the gluLookAt() call had placed the eye at (0.0, 0.0, 1.0), it would produce a distance of 1.0 to the origin. This isn't long enough to include the sphere in the view volume, and it would be clipped by the *zNear* clipping plane.

Similarly, if you place the eye at (0.0, 0.0, 10.0), the distance of 10.0 to the look at point will result in the unit sphere being 10.0 units away from the eye and far behind the *zFar* clipping plane placed at 7.0 units.

If this has confused you, read up on transformations in the OpenGL red book or OpenGL Specification. After you understand object coordinate space, eye coordinate space, and clip coordinate space, the above should become clear. Also, experiment with small test programs. If you're having trouble getting the correct transforms in your main application project, it can be educational to write a small piece of code that tries to reproduce the problem with simpler geometry.

8.090 How do I get a specified point (XYZ) to appear at the center of the scene?

gluLookAt() is the easiest way to do this. Simply set the X, Y, and Z values of your point as the fourth, fifth, and sixth parameters to gluLookAt().

8.100 I put my gluLookAt() call on my Projection matrix and now fog, lighting, and texture mapping don't work correctly. What happened?

Look at <u>question 8.030</u> for an explanation of this problem.

8.110 How can I create a stereo view?

Paul Bourke has assembled information on stereo OpenGL viewing.

- ◆ <u>3D Stereo Rendering Using OpenGL</u>
- ◆ <u>Calculating Stereo Pairs</u>
- ◆ Creating Anaglyphs using OpenGL

8 Using Viewing and Camera Transforms, and gluLookAt()

9 Transformations

9.001 I can't get transformations to work. Where can I learn more about matrices?

A thorough explanation of basic matrix math and linear algebra is beyond the scope of this FAQ. These concepts are taught in high school math classes in the United States.

If you understand the basics, but just get confused (a common problem even for the experienced!), read through Steve Baker's <u>review of matrix concepts</u> and his <u>article on Euler</u> <u>angles</u>.

Delphi code for performing basic vector, matrix, and quaternion operations can be found here.

9.005 Are OpenGL matrices column-major or row-major?

For programming purposes, OpenGL matrices are 16-value arrays with base vectors laid out contiguously in memory. The translation components occupy the 13th, 14th, and 15th elements of the 16-element matrix.

Column-major versus row-major is purely a notational convention. Note that post-multiplying with column-major matrices produces the same result as pre-multiplying with row-major matrices. The OpenGL Specification and the OpenGL Reference Manual both use column-major notation. You can use any notation, as long as it's clearly stated.

Sadly, the use of column-major format in the spec and blue book has resulted in endless confusion in the OpenGL programming community. Column-major notation suggests that matrices are not laid out in memory as a programmer would expect.

A summary of Usetnet postings on the subject can be found here.

9.010 What are OpenGL coordinate units?

The short answer: Anything you want them to be.

Depending on the contents of your geometry database, it may be convenient for your application to treat one OpenGL coordinate unit as being equal to one millimeter or one parsec or anything in between (or larger or smaller).

OpenGL also lets you specify your geometry with coordinates of differing values. For example, you may find it convenient to model an airplane's controls in centimeters, its fuselage in meters, and a world to fly around in kilometers. OpenGL's ModelView matrix can then scale these different coordinate systems into the same eye coordinate space.

It's the application's responsibility to ensure that the Projection and ModelView matrices are constructed to provide an image that keeps the viewer at an appropriate distance, with an appropriate field of view, and keeps the *zNear* and *zFar* clipping planes at an appropriate range. An application that displays molecules in micron scale, for example, would probably not want to place the viewer at a distance of 10 feet with a 60 degree field of view.

9.011 How are coordinates transformed? What are the different coordinate spaces?

9 Transformations

OpenGL FAQ and Troubleshooting Guide

Object Coordinates are transformed by the ModelView matrix to produce Eye Coordinates.

Eye Coordinates are transformed by the Projection matrix to produce Clip Coordinates.

Clip Coordinate X, Y, and Z are divided by Clip Coordinate W to produce Normalized Device Coordinates.

Normalized Device Coordinates are scaled and translated by the viewport parameters to produce Window Coordinates.

Object coordinates are the raw coordinates you submit to OpenGL with a call to glVertex*() or glVertexPointer(). They represent the coordinates of your object or other geometry you want to render.

Many programmers use a World Coordinate system. Objects are often modeled in one coordinate system, then scaled, translated, and rotated into the world you're constructing. World Coordinates result from transforming Object Coordinates by the modelling transforms stored in the ModelView matrix. However, OpenGL has no concept of World Coordinates. World Coordinates are purely an application construct.

Eye Coordinates result from transforming Object Coordinates by the ModelView matrix. The ModelView matrix contains both modelling and viewing transformations that place the viewer at the origin with the view direction aligned with the negative Z axis.

Clip Coordinates result from transforming Eye Coordinates by the Projection matrix. Clip Coordinate space ranges from $-W_c$ to W_c in all three axes, where W_c is the Clip Coordinate W value. OpenGL clips all coordinates outside this range.

Perspective division performed on the Clip Coordinates produces Normalized Device Coordinates, ranging from -1 to 1 in all three axes.

Window Coordinates result from scaling and translating Normalized Device Coordinates by the viewport. The parameters to glViewport() and glDepthRange() control this transformation. With the viewport, you can map the Normalized Device Coordinate cube to any location in your window and depth buffer.

For more information, see the **OpenGL Specification**, Figure 2.6.

9.020 How do I transform only one object in my scene or give each object its own transform?

OpenGL provides matrix stacks specifically for this purpose. In this case, use the ModelView matrix stack.

A typical OpenGL application first sets the matrix mode with a call to glMatrixMode(GL_MODELVIEW) and loads a viewing transform, perhaps with a call to gluLookAt().<u>More information is available on gluLookAt()</u>.

Then the code renders each object in the scene with its own transformation by wrapping the rendering with calls to glPushMatrix() and glPopMatrix(). For example:

```
glPushMatrix();
```

```
glRotatef(90., 1., 0., 0.);
gluCylinder(quad,1,1,2,36,12);
glPopMatrix();
```

The above code renders a cylinder rotated 90 degrees around the X-axis. The ModelView matrix is restored to its previous value after the glPopMatrix() call. Similar call sequences can render subsequent objects in the scene.

9.030 How do I draw 2D controls over my 3D rendering?

The basic strategy is to set up a 2D projection for drawing controls. You can do this either on top of your 3D rendering or in overlay planes. If you do so on top of a 3D rendering, you'll need to redraw the controls at the end of every frame (immediately before swapping buffers). If you draw into the overlay planes, you only need to redraw the controls if you're updating them.

To set up a 2D projection, you need to change the Projection matrix. Normally, it's convenient to set up the projection so one world coordinate unit is equal to one screen pixel, as follows:

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (0, windowWidth, 0, windowHeight);
```

gluOrtho2D() sets up a Z range of -1 to 1, so you need to use one of the glVertex2*() functions to ensure your geometry isn't clipped by the *zNear* or *zFar* clipping planes.

Normally, the ModelView matrix is set to the identity when drawing 2D controls, though you may find it convenient to do otherwise (for example, you can draw repeated controls with interleaved translation matrices).

If exact pixelization is required, you might want to put a small translation in the ModelView matrix, as shown below:

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.375, 0.375, 0.);
```

If you're drawing on top of a 3D–depth buffered image, you'll need to somehow disable depth testing while drawing your 2D geometry. You can do this by calling glDisable(GL_DEPTH_TEST) or glDepthFunc (GL_ALWAYS). Depending on your application, you might also simply clear the depth buffer before starting the 2D rendering. Finally, drawing all 2D geometry with a minimum Z coordinate is also a solution.

After the 2D projection is established as above, you can render normal OpenGL primitives to the screen, specifying their coordinates with XY pixel addresses (using OpenGL–centric screen coordinates, with (0,0) in the lower left).

9.040 How do I bypass OpenGL matrix transformations and send 2D coordinates directly for rasterization?

There isn't a mode switch to disable OpenGL matrix transformations. However, if you set either or both matrices to the identity with a glLoadIdentity() call, typical OpenGL implementations are intelligent enough to know that an identity transformation is a no-op and will act accordingly.

9 Transformations

OpenGL FAQ and Troubleshooting Guide

More detailed information on using OpenGL as a rasterization–only API is in the <u>OpenGL</u> <u>Game Developer's FAQ</u>.

9.050 What are the pros and cons of using absolute versus relative coordinates?

Some OpenGL applications may need to render the same object in multiple locations in a single scene. OpenGL lets you do this two ways:

1) Use "absolute coordinates". Maintain multiple copies of each object, each with its own unique set of vertices. You don't need to change the ModelView matrix to render the object at the desired location.

2) Use "relative coordinates". Keep only one copy of the object, and render it multiple times by pushing the ModelView matrix stack, setting the desired transform, sending the geometry, and popping the stack. Repeat these steps for each object.

In general, frequent changes to state, such as to the ModelView matrix, can negatively impact your application's performance. OpenGL can process your geometry faster if you don't wrap each individual primitive in a lot of changes to the ModelView matrix.

However, sometimes you need to weigh this against the memory savings of replicating geometry. Let's say you define a doorknob with high approximation, such as 200 or 300 triangles, and you're modeling a house with 50 doors in it, all of which have the same doorknob. It's probably preferable to use a single doorknob display list, with multiple unique transform matrices, rather than use absolute coordinates with 10–15K triangles in memory.

As with many computing issues, it's a trade–off between processing time and memory that you'll need to make on a case–by–case basis.

9.060 How can I draw more than one view of the same scene?

You can draw two views into the same window by using the glViewport() call. Set glViewport() to the area that you want the first view, set your scene's view, and render. Then set glViewport() to the area for the second view, again set your scene's view, and render.

You need to be aware that some operations don't pay attention to the glViewport, such as SwapBuffers and glClear(). SwapBuffers always swaps the entire window. However, you can restrain glClear() to a rectangular window by using the scissor rectangle.

Your application might only allow different views in separate windows. If so, you need to perform a MakeCurrent operation between the two renderings. If the two windows share a context, you need to change the scene's view as described above. This might not be necessary if your application uses separate contexts for each window.

9.070 How do I transform my objects around a fixed coordinate system rather than the object's local coordinate system?

If you rotate an object around its Y-axis, you'll find that the X- and Z-axes rotate with the object. A subsequent rotation around one of these axes rotates around the newly transformed axis and not the original axis. It's often desirable to perform transformations in a fixed coordinate system rather than the object's local coordinate system.

The <u>OpenGL Game Developer's FAO</u> contains information on using quaternions to store rotations, which may be useful in solving this problem.

The root cause of the problem is that OpenGL matrix operations postmultiply onto the matrix stack, thus causing transformations to occur in object space. To affect screen space transformations, you need to premultiply. OpenGL doesn't provide a mode switch for the order of matrix multiplication, so you need to premultiply by hand. An application might implement this by retrieving the current matrix after each frame. The application multiplies new transformations for the next frame on top of an identity matrix and multiplies the accumulated current transformations (from the last frame) onto those transformations using glMultMatrix().

You need to be aware that retrieving the ModelView matrix once per frame might have a detrimental impact on your application's performance. However, you need to benchmark this operation, because the performance will vary from one implementation to the next.

9.080 What are the pros and cons of using glFrustum() versus gluPerspective()? Why would I want to use one over the other?

glFrustum() and gluPerspective() both produce perspective projection matrices that you can use to transform from eye coordinate space to clip coordinate space. The primary difference between the two is that glFrustum() is more general and allows off–axis projections, while gluPerspective() only produces symmetrical (on–axis) projections. Indeed, you can use glFrustum() to implement gluPerspective(). However, aside from the layering of function calls that is a natural part of the GLU interface, there is no performance advantage to using matrices generated by glFrustum() over gluPerspective().

Since glFrustum() is more general than gluPerspective(), you can use it in cases when gluPerspective() can't be used. Some examples include <u>projection shadows</u>, tiled renderings, and stereo views.

Tiled rendering uses multiple off-axis projections to render different sections of a scene. The results are assembled into one large image array to produce the final image. This is often necessary when the desired dimensions of the final rendering exceed the OpenGL implementation's maximum viewport size.

In a stereo view, two renderings of the same scene are done with the view location slightly shifted. Since the view axis is right between the "eyes", each view must use a slightly off–axis projection to either side to achieve correct visual results.

9.085 How can I make a call to glFrustum() that matches my call to gluPerspective()?

The field of view (fov) of your glFrustum() call is:

fov*0.5 = arctan ((top-bottom)*0.5 / near)

Since *bottom* == -top for the symmetrical projection that gluPerspective() produces, then:

top = tan(fov*0.5) * nearbottom = -top The *left* and *right* parameters are simply functions of the *top*, *bottom*, and *aspect*:

left = aspect * bottom
right = aspect * top

The OpenGL Reference Manual (where do I get this?) shows the matrices produced by both functions.

9.090 How do I draw a full-screen quad?

This question usually means, "How do I draw a quad that fills the entire OpenGL viewport?" There are many ways to do this.

The most straightforward method is to set the desired color, set both the Projection and ModelView matrices to the identity, and call glRectf() or draw an equivalent GL_QUADS primitive. Your rectangle or quad's Z value should be in the range of -1.0 to 1.0, with -1.0 mapping to the *zNear* clipping plane, and 1.0 to the *zFar* clipping plane.

As an example, here's how to draw a full-screen quad at the *zNear* clipping plane:

```
glMatrixMode (GL_MODELVIEW);
glPushMatrix ();
glLoadIdentity ();
glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
glBegin (GL_QUADS);
glVertex3i (-1, -1, -1);
glVertex3i (1, -1, -1);
glVertex3i (1, 1, -1);
glVertex3i (-1, 1, -1);
glEnd ();
glPopMatrix ();
glPopMatrix ();
```

Your application might want the quad to have a maximum Z value, in which case 1 should be used for the Z value instead of -1.

When painting a full–screen quad, it might be useful to mask off some buffers so that only specified buffers are touched. For example, you might mask off the color buffer and set the depth function to GL_ALWAYS, so only the depth buffer is painted. Also, you can set masks to allow the stencil buffer to be set or any combination of buffers.

9.100 How can I find the screen coordinates for a given object-space coordinate?

You can use the GLU library gluProject() utility routine if you only need to find it for a few vertices. For a large number of coordinates, it can be more efficient to use the Feedback mechanism.

To use gluProject(), you'll need to provide the ModelView matrix, projection matrix, viewport, and input object space coordinates. Screen space coordinates are returned for X, Y, and Z, with Z being normalized ($0 \le Z \le 1$).

9.110 How can I find the object-space coordinates for a pixel on the screen?

The GLU library provides the gluUnProject() function for this purpose.

You'll need to read the depth buffer to obtain the input screen coordinate Z value at the X,Y location of interest. This can be coded as follows:

GLdouble z;

glReadPixels (x, y, 1, 1, GL_DEPTH_COMPONENT, GL_DOUBLE, &z);

Note that x and y are OpenGL–centric with (0,0) in the lower–left corner.

You'll need to provide the screen space X, Y, and Z values as input to gluUnProject() with the ModelView matrix, Projection matrix, and viewport that were current at the time the specific pixel of interest was rendered.

9.120 How do I find the coordinates of a vertex transformed only by the ModelView matrix?

It's often useful to obtain the eye coordinate space value of a vertex (i.e., the object space vertex transformed by the ModelView matrix). You can obtain this by retrieving the current ModelView matrix and performing simple vector / matrix multiplication.

9.130 How do I calculate the object-space distance from the viewer to a given point?

Transform the point into eye–coordinate space by multiplying it by the ModelView matrix. Then simply calculate its distance from the origin. (If this doesn't work, you may have incorrectly placed the view transform on the Projection matrix stack.)

9.140 How do I keep my aspect ratio correct after a window resize?

It depends on how you are setting your projection matrix. In any case, you'll need to know the new dimensions (width and height) of your window. How to obtain these depends on which platform you're using. In GLUT, for example, the dimensions are passed as parameters to the reshape function callback.

The following assumes you're maintaining a viewport that's the same size as your window. If you are not, substitute viewportWidth and viewportHeight for windowWidth and windowHeight.

If you're using gluPerspective() to set your Projection matrix, the second parameter controls the aspect ratio. When your program catches a window resize, you'll need to change your Projection matrix as follows:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fov, (float)windowWidth/(float)windowHeight, zNear, zFar);
```

If you're using glFrustum(), the aspect ratio varies with the width of the view volume to the height of the view volume. You might maintain a 1:1 aspect ratio with the following window resize code:

float cx, halfWidth = windowWidth*0.5f;

```
float aspect = (float)windowWidth/(float)windowHeight;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
/* cx is the eye space center of the zNear plane in X */
glFrustum(cx-halfWidth*aspect, cx+halfWidth*aspect, bottom, top, zNear, zFar);
```

glOrtho() and gluOrtho2D() are similar to glFrustum().

9.150 Can I make OpenGL use a left-handed coordinate space?

OpenGL doesn't have a mode switch to change from right– to left–handed coordinates. However, you can easily obtain a left–handed coordinate system by multiplying a negative Z scale onto the ModelView matrix. For example:

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glScalef (1., 1., -1.);
/* multiply view transforms as usual... */
/* multiply model transforms as usual... */
```

9.160 How can I transform an object so that it points at or follows another object or point in my scene?

You need to construct a matrix that transforms from your object's local coordinate system into a coordinate system that faces in the desired direction. <u>See this example code</u> to see how this type of matrix is created.

If you merely want to render an object so that it always faces the viewer, you might consider simply rendering it in eye–coordinate space with the ModelView matrix set to the identity.

9.162 How can I transform an object with a given yaw, pitch, and roll?

The upper left 3x3 portion of a transformation matrix is composed of the new X, Y, and Z axes of the post–transformation coordinate space.

If the new transform is a roll, compute new local Y and X axes by rotating them "roll" degrees around the local Z axis. Do similar calculations if the transform is a pitch or yaw. Then simply construct your transformation matrix by inserting the new local X, Y, and Z axes into the upper left 3x3 portion of an identity matrix. This matrix can be passed as a parameter to glMultMatrix().

Further rotations should be computed around the new local axes. This will inevitably require rotation about an arbitrary axis, which can be confusing to inexperienced 3D programmers. This is a <u>basic concept in linear algebra</u>.

Many programmers apply all three transformations -- yaw, pitch, and roll -- at once as successive glRotate() calls about the X, Y, and Z axes. This has the disadvantage of creating gimbal lock, in which the result depends on the order of glRotate() calls.

9.170 How do I render a mirror?

Render your scene twice, once as it is reflected in the mirror, then once from the normal (non–reflected) view. Example code demonstrates this technique.

9 Transformations

OpenGL FAQ and Troubleshooting Guide

For axis–aligned mirrors, such as a mirror on the YZ plane, the reflected scene can be rendered with a simple scale and translate. Scale by -1.0 in the axis corresponding to the mirror's normal, and translate by twice the mirror's distance from the origin. Rendering the scene with these transforms in place will yield the scene reflected in the mirror. Use the matrix stack to restore the view transform to its previous value.

Next, clear the depth buffer with a call to glClear(GL_DEPTH_BUFFER_BIT). Then render the mirror. For a perfectly reflecting mirror, render into the depth buffer only. Real mirrors are not perfect reflectors, as they absorb some light. To create this effect, use blending to render a black mirror with an alpha of 0.05. glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA) is a good blending

function for this purpose.

Finally, render the non–reflected scene. Since the entire reflected scene exists in the color buffer, and not just the portion of the reflected scene in the mirror, you will need to touch all pixels to overwrite areas of the reflected scene that should not be visible.

9.180 How can I do my own perspective scaling?

OpenGL multiplies your coordinates by the ModelView matrix, then by the Projection matrix to get clip coordinates. It then performs the perspective divide to obtain normalized device coordinates. It's the perspective division step that creates a perspective rendering, with geometry in the distance appearing smaller than the geometry in the foreground. The perspective division stage is accomplished by dividing your XYZ clipping coordinate values by the clipping coordinate W value, such as:

Xndc = Xcc/Wcc Yndc = Ycc/Wcc Zndc = Zcc/Wcc

To do your own perspective correction, you need to obtain the clipping coordinate W value. The feedback buffer provides homogenous coordinates with XYZ in device coordinates and W in clip coordinates. You might also glGetFloatv(GL_CURRENT_RASTER_POSITION,...) and the W value will again be in clipping coordinates, while XYZ are in device coordinates.

10 Clipping, Culling, and Visibility Testing

10.010 How do I tell if a vertex has been clipped or not?

You can use the OpenGL Feedback feature to determine if a vertex will be clipped or not. After you're in Feedback mode, simply send the vertex in question as a GL_POINTS primitive. Then switch back to GL_RENDER mode and check the size of the Feedback buffer. A size of zero indicates a clipped vertex.

Typically, OpenGL implementations don't provide a fast feedback mechanism. It might be faster to perform the clip test manually. To do so, construct six plane equations corresponding to the clip–coordinate view volume and transform them into object space by the current ModelView matrix. A point is clipped if it violates any of the six plane equations.

Here's a <u>GLUT example</u> that shows how to calculate the object–space view–volume planes and clip test bounding boxes against them.

10.020 How do I perform occlusion or visibility testing?

OpenGL provides no direct support for determining whether a given primitive will be visible in a scene for a given viewpoint. At worst, an application will need to perform these tests manually. <u>The previous question contains information on how to do this</u>.

The code example from question 10.010 was combined with Nate Robins' excellent viewing tutorial to produce <u>this view culling example code</u>.

Higher-level APIs, such as Fahernheit Large Model, may provide this feature.

HP OpenGL platforms support an Occlusion Culling extension. To use this extension, enable the occlusion test, render some bounding geometry, and call glGetBooleanv() to obtain the visibility status of the geometry.

10.030 How do I render to a nonrectangular viewport?

OpenGL's stencil buffer can be used to mask the area outside of a non-rectangular viewport. With stencil enabled and stencil test appropriately set, rendering can then occur in the unmasked area. Typically an application will write the stencil mask once, and then render repeated frames into the unmasked area.

As with the depth buffer, an application must ask for a stencil buffer when the window and context are created.

An application will perform such a rendering as follows:

/* Enable stencil test and leave it enabled throughout */
glEnable (GL_STENCIL_TEST);

/* Prepare to write a single bit into the stencil buffer in the area outsi glStencilFunc (GL_ALWAYS, 0x1, 0x1);

/* Render a set of geometry corresponding to the area outside the viewport

/* The stencil buffer now has a single bit painted in the area outside the
/* Prepare to render the scene in the viewport */
glStencilFunc (GL_EQUAL, 0x0, 0x1);
/* Render the scene inside the viewport here */
/* ...render the scene again as needed for animation purposes */

After a single bit is painted in the area outside the viewport, an application may render geometry to either the area inside or outside the viewport. To render to the inside area, use glStencilFunc(GL_EQUAL,0x0,0x1), as the code above shows. To render to the area outside the viewport, use glStencilFunc(GL_EQUAL,0x1,0x1).

You can obtain similar results using only the depth test. After rendering a 3D scene to a rectangular viewport, an app can clear the depth buffer and render the nonrectangular frame.

10.040 When an OpenGL primitive moves placing one vertex outside the window, suddenly the color or texture mapping is incorrect. What's going on?

There are two potential causes for this.

When a primitive lies partially outside the window, it often crosses the view volume boundary. OpenGL must clip any primitive that crosses the view volume boundary. To clip a primitive, OpenGL must interpolate the color values, so they're correct at the new clip vertex. This interpolation is perspective correct. However, when a primitive is rasterized, the color values are often generated using linear interpolation in window space, which isn't perspective correct. The difference in generated color values means that for any given barycentric coordinate location on a filled primitive, the color values may be different depending on whether the primitive is clipped. If the color values generated during rasterization were perspective correct, this problem wouldn't exist.

For some OpenGL implementations, texture coordinates generated during rasterization aren't perspective correct. However, you can usually make them perspective correct by calling glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);. Colors generated at the rasterization stage aren't perspective correct in almost every OpenGL implementation, and can't be made so. For this reason, you're more likely to encounter this problem with colors than texture coordinates.

A second reason the color or texture mapping might be incorrect for a clipped primitive is because the color values or texture coordinates are nonplanar. Color values are nonplanar when the three color components at each vertex don't lie in a plane in 3D color space. 2D texture coordinates are always planar. However, in this context, the term nonplanar is used for texture coordinates that look up a texel area that isn't congruent in shape to the primitive being textured.

Nonplanar colors or texture coordinates aren't a problem for triangular primitives, but the problem may occur with GL_QUADS, GL_QUAD_STRIP and GL_POLYGON primitives. When using nonplanar color values or texture coordinates, there isn't a correct way to generate new values associated with clipped vertices. Even perspective–correct interpolation can create differences between clipped and nonclipped primitives. The solution to this problem is to not use nonplanar color values and texture coordinates.

10.050 I know my geometry is inside the view volume. How can I turn off OpenGL's view-volume clipping to maximize performance?

Standard OpenGL doesn't provide a mechanism to disable the view–volume clipping test; thus, it will occur for every primitive you send.

Some implementations of OpenGL support the GL_EXT_clip_volume_hint extension. If the extension is available, a call to

glHint(GL_CLIP_VOLUME_CLIPPING_HINT_EXT,GL_FASTEST) will inform OpenGL that the geometry is entirely within the view volume and that view–volume clipping is unnecessary. Normal clipping can be resumed by setting this hint to GL_DONT_CARE. When clipping is disabled with this hint, results are undefined if geometry actually falls outside the view volume.

10.060 When I move the viewpoint close to an object, it starts to disappear. How can I disable OpenGL's zNear clipping plane?

You can't. If you think about it, it makes sense: What if the viewpoint is in the middle of a scene? Certainly some geometry is behind the viewer and needs to be clipped. Rendering it will produce undesirable results.

For correct perspective and depth buffer calculations to occur, setting the *zNear* clipping plane to 0.0 is also not an option. The *zNear* clipping plane must be set at a positive (nonzero) distance in front of the eye.

To avoid the clipping artifacts that can otherwise occur, an application must track the viewpoint location within the scene, and ensure it doesn't get too close to any geometry. You can usually do this with a simple form of collision detection. This FAQ contains more information on collision detection with OpenGL.

If you're certain that your geometry doesn't intersect any of the view–volume planes, you might be able to use an extension to disable clipping. See <u>the previous question</u> for more information.

10.070 How do I draw glBitmap() or glDrawPixels() primitives that have an initial glRasterPos() outside the window's left or bottom edge?

When the raster position is set outside the window, it's often outside the view volume and subsequently marked as invalid. Rendering the glBitmap and glDrawPixels primitives won't occur with an invalid raster position. Because glBitmap/glDrawPixels produce pixels up and to the right of the raster position, it appears impossible to render this type of primitive clipped by the left and/or bottom edges of the window.

However, here's an often-used trick: Set the raster position to a valid value inside the view volume. Then make the following call:

glBitmap (0, 0, 0, 0, xMove, yMove, NULL);

This tells OpenGL to render a no–op bitmap, but move the current raster position by (xMove, yMove). Your application will supply (xMove, yMove) values that place the raster position outside the view volume. Follow this call with the glBitmap() or glDrawPixels() to

do the rendering you desire.

10.080 Why doesn't glClear() work for areas outside the scissor rectangle?

The OpenGL Specification states that glClear() only clears the scissor rectangle when the scissor test is enabled. If you want to clear the entire window, use the code:

glDisable (GL_SCISSOR_TEST); glClear (...); glEnable (GL_SCISSOR_TEST);

10.090 How does face culling work? Why doesn't it use the surface normal?

OpenGL face culling calculates the signed area of the filled primitive in window coordinate space. The signed area is positive when the window coordinates are in a counter–clockwise order and negative when clockwise. An app can use glFrontFace() to specify the ordering, counter–clockwise or clockwise, to be interpreted as a front–facing or back–facing primitive. An application can specify culling either front or back faces by calling glCullFace(). Finally, face culling must be enabled with a call to glEnable(GL_CULL_FACE); .

OpenGL uses your primitive's window space projection to determine face culling for two reasons. To create interesting lighting effects, it's often desirable to specify normals that aren't orthogonal to the surface being approximated. If these normals were used for face culling, it might cause some primitives to be culled erroneously. Also, a dot-product culling scheme could require a matrix inversion, which isn't always possible (i.e., in the case where the matrix is singular), whereas the signed area in DC space is always defined.

However, some OpenGL implementations support the GL_EXT_ cull_vertex extension. If this extension is present, an application may specify a homogeneous eye position in object space. Vertices are flagged as culled, based on the dot product of the current normal with a vector from the vertex to the eye. If all vertices of a primitive are culled, the primitive isn't rendered. In many circumstances, using this extension results in faster rendering, because it culls faces at an earlier stage of the rendering pipeline.

11 Color

11.010 My texture map colors reverse blue and red, yellow and cyan, etc. What's happening?

Your texture image has the reverse byte ordering of what OpenGL is expecting. One way to handle this is to swap bytes within your code before passing the data to OpenGL.

Under OpenGL 1.2, you may specify GL_BGR or GL_BGRA as the "format" parameter to glDrawPixels(), glGetTexImage(), glReadPixels(), glTexImage1D(), glTexImage2D(), and glTexImage3D(). In previous versions of OpenGL, this functionality might be available in the form of the EXT_bgra extension (using GL_BGR_EXT and GL_BGRA_EXT as the "format" parameter).

11.020 How do I render a color index into an RGB window or vice versa?

There isn't a way to do this. However, you might consider opening an RGB window with a color index overlay plane, if it works in your application.

If you have an array of color indices that you want to use as a texture map, you might want to consider using GL_EXT_paletted_texture, which lets an application specify a color index texture map with a color palette.

11.030 The colors are almost entirely missing when I render in Microsoft Windows. What's happening?

The most probable cause is that the Windows display is set to 256 colors. To change it, you can increase the color depth by clicking the right mouse button on the desktop, then select Properties, the Settings tab, and change the number of colors in the Color Palette to a higher number.

11.040 How do I specify an exact color for a primitive?

First, you'll need to know the depth of the color buffer you are rendering to. For an RGB color buffer, you can obtain these values with the following code:

GLint redBits, greenBits, blueBits; glGetIntegerv (GL_RED_BITS, &redBits); glGetIntegerv (GL_GREEN_BITS, &greenBits); glGetIntegerv (GL_BLUE_BITS, &blueBits);

If the depth value for each component is at least as large as your required color precision, you can specify an exact color for your primitives. Specify the color you want to use into the most significant bits of three unsigned integers and use glColor3ui() to specify the color.

If your color buffer isn't deep enough to accurately represent the color you desire, you'll need a fallback strategy. Trimming off the least significant bits of each color component is an acceptable alternative. Again, use glColor3ui() (or glColor3us(), etc.) to specify the color with your values stored in the most significant bits of each parameter.

In either event, you'll need to ensure that any state that could affect the final color has been disabled. The following code will accomplish this:

```
glDisable (GL_BLEND);
glDisable (GL_DITHER);
glDisable (GL_FOG);
glDisable (GL_LIGHTING);
glDisable (GL_TEXTURE_1D);
glDisable (GL_TEXTURE_2D);
glDisable (GL_TEXTURE_3D);
glShadeModel (GL_FLAT);
```

11.050 How do I render each primitive in a unique color?

You need to know the depth of each component in your color buffer. The previous question contains the code to obtain these values. The depth tells you the number of unique color values you can render. For example, if you use the code from the previous question, which retrieves the color depth in redBits, greenBits, and blueBits, the number of unique colors available is 2^(redBits+greenBits+blueBits).

If this number is greater than the number of primitives you want to render, there is no problem. You need to use glColor3ui() (or glColor3us(), etc) to specify each color, and store the desired color in the most significant bits of each parameter. You can code a loop to render each primitive in a unique color with the following:

```
/*
   Given: numPrims is the number of primitives to render.
   Given void renderPrimitive(unsigned long) is a routine to render the pr
   Given GLuint makeMask (GLint) returns a bit mask for the number of bits
 * /
GLuint redMask = makeMask(redBits) << (greenBits + blueBits);
GLuint greenMask = makeMask(greenBits) << blueBits;
GLuint blueMask = makeMask(blueBits);
int redShift = 32 - (redBits+greenBits+blueBits);
int greenShift = 32 - (greenBits+blueBits);
int blueShift = 32 - blueBits;
unsigned long indx;
for (indx=0; indx<numPrims, indx++) {</pre>
   glColor3ui (indx & redMask << redShift,
               indx & greenMask << greenShift,</pre>
               indx & blueMask << blueShift);</pre>
   renderPrimitive (indx);
}
```

Also, make sure you disable any state that could alter the final color. <u>See the question</u> above for a code snippet to accomplish this.

If you're using this for picking instead of the usual Selection feature, any color subsequently read back from the color buffer can easily be converted to the indx value of the primitive rendered in that color.

12 The Depth Buffer

12.010 How do I make depth buffering work?

Your application needs to do at least the following to get depth buffering to work:

- 1. Ask for a depth buffer when you create your window.
- 2. Place a call to glEnable (GL_DEPTH_TEST) in your program's initialization routine, after a context is created and made current.
- 3. Ensure that your *zNear* and *zFar* clipping planes are set correctly and in a way that provides adequate depth buffer precision.
- 4. Pass GL_DEPTH_BUFFER_BIT as a parameter to glClear, typically bitwise OR'd with other values such as GL_COLOR_BUFFER_BIT.

There are a number of OpenGL example programs available on the Web, which use depth buffering. If you're having trouble getting depth buffering to work correctly, you might benefit from looking at an example program to see what is done differently. This FAQ contains links to several web sites that have example OpenGL code.

12.020 Depth buffering doesn't work in my perspective rendering. What's going on?

Make sure the *zNear* and *zFar* clipping planes are specified correctly in your calls to glFrustum() or gluPerspective().

A mistake many programmers make is to specify a *zNear* clipping plane value of 0.0 or a negative value which isn't allowed. Both the *zNear* and *zFar* clipping planes are positive (not zero or negative) values that represent distances in front of the eye.

Specifying a *zNear* clipping plane value of 0.0 to gluPerspective() won't generate an OpenGL error, but it might cause depth buffering to act as if it's disabled. A negative *zNear* or *zFar* clipping plane value would produce undesirable results.

A *zNear* or *zFar* clipping plane value of zero or negative, when passed to glFrustum(), will produce an error that you can retrieve by calling glGetError(). The function will then act as a no–op.

12.030 How do I write a previously stored depth image to the depth buffer?

Use the glDrawPixels() command, with the format parameter set to GL_DEPTH_COMPONENT. You may want to mask off the color buffer when you do this, with a call to glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE); .

12.040 Depth buffering seems to work, but polygons seem to bleed through polygons that are in front of them. What's going on?

You may have configured your *zNear* and *zFar* clipping planes in a way that severely limits your depth buffer precision. Generally, this is caused by a *zNear* clipping plane value that's too close to 0.0. As the *zNear* clipping plane is set increasingly closer to 0.0, the effective precision of the depth buffer decreases dramatically. Moving the *zFar* clipping plane further away from the eye always has a negative impact on depth buffer precision, but it's not one as

OpenGL FAQ and Troubleshooting Guide

dramatic as moving the *zNear* clipping plane.

The <u>OpenGL Reference Manual</u> description for glFrustum() relates depth precision to the *zNear* and *zFar* clipping planes by saying that roughly log2(*zFar/zNear*) bits of precision are lost. Clearly, as *zNear* approaches zero, this equation approaches infinity.

While the blue book description is good at pointing out the relationship, it's somewhat inaccurate. As the ratio (zFar/zNear) increases, less precision is available near the back of the depth buffer and more precision is available close to the front of the depth buffer. So primitives are more likely to interact in Z if they are further from the viewer.

It's possible that you simply don't have enough precision in your depth buffer to render your scene. See <u>the last question in this section</u> for more info.

It's also possible that you are drawing coplanar primitives. Round–off errors or differences in rasterization typically create "Z fighting" for coplanar primitives. Here are some options to assist you when rendering coplanar primitives.

12.050 Why is my depth buffer precision so poor?

The depth buffer precision in eye coordinates is strongly affected by the ratio of *zFar* to *zNear*, the *zFar* clipping plane, and how far an object is from the *zNear* clipping plane.

You need to do whatever you can to push the *zNear* clipping plane out and pull the *zFar* plane in as much as possible.

To be more specific, consider the transformation of depth from eye coordinates

 x_e, y_e, z_e, w_e

to window coordinates

 x_w, y_w, z_w

with a perspective projection matrix specified by

glFrustum(l, r, b, t, n, f);

and assume the default viewport transform. The clip coordinates of z_c and w_c are

$$z_c = -z_e^* (f+n)/(f-n) - w_e^* 2^* f^*n/(f-n)$$

 $w_c = -z_e$

Why the negations? OpenGL wants to present to the programmer a right-handed coordinate system before projection and left-handed coordinate system after projection.

and the ndc coordinate:

 $z_{ndc} = z_c / w_c = [-z_e * (f+n)/(f-n) - w_e * 2*f*n/(f-n)] / -z_e$

$$= (f+n)/(f-n) + (w_e / z_e) * 2*f*n/(f-n)$$

The viewport transformation scales and offsets by the depth range (Assume it to be [0, 1]) and then scales by $s = (2^n-1)$ where n is the bit depth of the depth buffer:

$$z_w = s * [(w_e / z_e) * f*n/(f-n) + 0.5 * (f+n)/(f-n) + 0.5]$$

Let's rearrange this equation to express z_e / w_e as a function of z_w

$$z_e / w_e = f*n/(f-n) / ((z_w / s) - 0.5 * (f+n)/(f-n) - 0.5)$$
$$= f*n / ((z_w / s) * (f-n) - 0.5 * (f+n) - 0.5 * (f-n))$$
$$= f*n / ((z_w / s) * (f-n) - f) [*]$$

Now let's look at two points, the *zNear* clipping plane and the *zFar* clipping plane:

$$\begin{split} z_w &= 0 \Longrightarrow z_e \; / \; w_e = f \, \ast \, n \; / \; (-f) = -n \\ z_w &= s \Longrightarrow z_e \; / \; w_e = f \, \ast \, n \; / \; ((f{-}n) - f) = -f \end{split}$$

In a fixed-point depth buffer, z_w is quantized to integers. The next representable z buffer depth away from the clip planes are 1 and s-1:

$$z_w = 1 \Longrightarrow z_e / w_e = f * n / ((1/s) * (f-n) - f)$$

$$z_w = s-1 \Longrightarrow z_e / w_e = f * n / (((s-1)/s) * (f-n) - f)$$

Now let's plug in some numbers, for example, n = 0.01, f = 1000 and s = 65535 (i.e., a 16-bit depth buffer)

$$z_w = 1 \Longrightarrow z_e / w_e = -0.01000015$$

 $z_w = s - 1 \Longrightarrow z_e / w_e = -395.90054$

Think about this last line. Everything at eye coordinate depths from -395.9 to -1000 has to map into either 65534 or 65535 in the z buffer. Almost two thirds of the distance between the *zNear* and *zFar* clipping planes will have one of two z–buffer values!

To further analyze the z-buffer resolution, let's take the derivative of [*] with respect to z_w

$$d(z_e / w_e) / dz_w = -f * n * (f-n) * (1/s) / ((z_w / s) * (f-n) - f)^2$$

Now evaluate it at $z_w = s$

$$d (z_e / w_e) / d z_w = - f * (f-n) * (1/s) / n$$
$$= - f * (f/n-1) / s [**]$$

If you want your depth buffer to be useful near the *zFar* clipping plane, you need to keep this value to less than the size of your objects in eye space (for most practical uses, world space).

12.060 How do I turn off the zNear clipping plane?

See this question in the Clipping section.

12.070 Why is there more precision at the front of the depth buffer?

After the projection matrix transforms the clip coordinates, the XYZ–vertex values are divided by their clip coordinate W value, which results in normalized device coordinates. This step is known as the perspective divide. The clip coordinate W value represents the distance from the eye. As the distance from the eye increases, 1/W approaches 0. Therefore, X/W and Y/W also approach zero, causing the rendered primitives to occupy less screen space and appear smaller. This is how computers simulate a perspective view.

As in reality, motion toward or away from the eye has a less profound effect for objects that are already in the distance. For example, if you move six inches closer to the computer screen in front of your face, it's apparent size should increase quite dramatically. On the other hand, if the computer screen were already 20 feet away from you, moving six inches closer would have little noticeable impact on its apparent size. The perspective divide takes this into account.

As part of the perspective divide, Z is also divided by W with the same results. For objects that are already close to the back of the view volume, a change in distance of one coordinate unit has less impact on Z/W than if the object is near the front of the view volume. To put it another way, an object coordinate Z unit occupies a larger slice of NDC–depth space close to the front of the view volume than it does near the back of the view volume.

In summary, the perspective divide, by its nature, causes more Z precision close to the front of the view volume than near the back.

A previous question in this section contains related information.

12.080 There is no way that a standard-sized depth buffer will have enough precision for my astronomically large scene. What are my options?

The typical approach is to use a multipass technique. The application might divide the geometry database into regions that don't interfere with each other in Z. The geometry in each region is then rendered, starting at the furthest region, with a clear of the depth buffer before each region is rendered. This way the precision of the entire depth buffer is made available to each region.

13 Drawing Lines over Polygons and Using Polygon Offset

13.010 What are the basics for using polygon offset?

It's difficult to render coplanar primitives in OpenGL for two reasons:

- Given two overlapping coplanar primitives with different vertices, floating point round-off errors from the two polygons can generate different depth values for overlapping pixels. With depth test enabled, some of the second polygon's pixels will pass the depth test, while some will fail.
- For coplanar lines and polygons, vastly different depth values for common pixels can result. This is because depth values from polygon rasterization derive from the polygon's plane equation, while depth values from line rasterization derive from linear interpolation.

Setting the depth function to GL_LEQUAL or GL_EQUAL won't resolve the problem. The visual result is referred to as *stitching, bleeding,* or *Z fighting*.

Polygon offset was an extension to OpenGL 1.0, and is now incorporated into OpenGL 1.1. It allows an application to define a depth offset, which can apply to filled primitives, and under OpenGL 1.1, it can be separately enabled or disabled depending on whether the primitives are rendered in fill, line, or point mode. Thus, an application can render coplanar primitives by first rendering one primitive, then by applying an offset and rendering the second primitive.

While polygon offset can alter the depth value of filled primitives in point and line mode, under no circumstances will polygon offset affect the depth values of GL_POINTS, GL_LINES, GL_LINE_STRIP, or GL_LINE_LOOP primitives. If you are trying to render point or line primitives over filled primitives, use polygon offset to push the filled primitives back. (It can't be used to pull the point and line primitives forward.)

Because polygon offset alters the correct Z value calculated during rasterization, the resulting Z value, which is stored in the depth buffer will contain this offset and can adversely affect the resulting image. In many circumstances, undesirable "bleed-through" effects can result. Indeed, polygon offset may cause some primitives to pass the depth test entirely when they normally would not, or vice versa. When models intersect, polygon offset can cause an inaccurate rendering of the intersection point.

13.020 What are the two parameters in a glPolygonOffset() call and what do they mean?

Polygon offset allows the application to specify a depth offset with two parameters, *factor* and *units. factor* scales the maximum Z slope, with respect to X or Y of the polygon, and *units* scales the minimum resolvable depth buffer value. The results are summed to produce the depth offset. This offset is applied in screen space, typically with positive Z pointing into the screen.

The *factor* parameter is required to ensure correct results for filled primitives that are nearly edge–on to the viewer. In this case, the difference between Z values for the same pixel

generated by two coplanar primitives can be as great as the maximum Z slope in X or Y. This Z slope will be large for nearly edge–on primitives, and almost non–existent for face–on primitives. The *factor* parameter lets you add this type of variable difference into the resulting depth offset.

A typical use might be to set *factor* and *units* to 1.0 to offset primitives into positive Z (into the screen) and enable polygon offset for fill mode. Two passes are then made, once with the model's solid geometry and once again with the line geometry. Nearly edge–on filled polygons are pushed substantially away from the eyepoint, to minimize interference with the line geometry, while nearly planar polygons are drawn at least one depth buffer unit behind the line geometry.

13.030 What's the difference between the OpenGL 1.0 polygon offset extension and OpenGL 1.1 (and later) polygon offset interfaces?

The 1.0 polygon offset extension didn't let you apply the offset to filled primitives in line or point mode. Only filled primitives in fill mode could be offset.

In the 1.0 extension, a *bias* parameter was added to the normalized (0.0 - 1.0) depth value, in place of the 1.1 *units* parameter. Typical applications might obtain a good offset by specifying a *bias* of 0.001.

See the <u>GLUT example</u>, which renders two cylinders, one using the 1.0 polygon offset extension and the other using the 1.1 polygon offset interface.

13.040 Why doesn't polygon offset work when I draw line primitives over filled primitives?

Polygon offset, as its name implies, only works with polygonal primitives. It affects only the filled primitives: GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, and GL_POLYGON. Polygon offset will work when you render them with glPolygonMode set to GL_FILL, GL_LINE, or GL_POINT.

Polygon offset doesn't affect non-polygonal primitives. The GL_POINTS, GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP primitives can't be offset with glPolygonOffset().

13.050 What other options do I have for drawing coplanar primitives when I don't want to use polygon offset?

You can simulate the effects of polygon offset by tinkering with glDepthRange(). For example, you might code the following:

glDepthRange (0.1, 1.0);
/* Draw underlying geometry */
glDepthRange (0.0, 0.9);
/* Draw overlying geometry */

This code provides a fixed offset in Z, but doesn't account for the polygon slope. It's roughly equivalent to using glPolygonOffset with a factor parameter of 0.0.

You can render coplanar primitives with the Stencil buffer in many creative ways. The <u>OpenGL</u> <u>Programming Guide</u> outlines one well–know method. The algorithm for drawing a polygon and its outline is as follows:

13 Drawing Lines over Polygons and Using Polygon Offset

- 1. Draw the outline into the color, depth, and stencil buffers.
- 2. Draw the filled primitive into the color buffer and depth buffer, but only where the stencil buffer is clear.
- 3. Mask off the color and depth buffers, and render the outline to clear the stencil buffer.

On some SGI OpenGL platforms, an application can use the SGIX_reference_plane extension. With this extension, the user specifies a plane equation in object coordinates corresponding to a set of coplanar primitives. You can enable or disable the plane. When the plane is enabled, all fragment Z values will derive from the specified plane equation. Thus, for any given fragment XY location, the depth value is guaranteed to be identical regardless of which primitive rendered it.

14 Rasterization and Operations on the Framebuffer

14.010 How do I obtain the address of the OpenGL framebuffer, so I can write directly to it?

OpenGL doesn't provide a standard mechanism to let an application obtain the address of the framebuffer. If an implementation allows this, it's through an extension.

Typically, programmers who write graphics programs for a single standard graphics hardware format, such as the VGA standard under Microsoft Windows, will want the framebuffer's address. The programmers need to understand that OpenGL is designed to run on a wide variety of graphics hardware, many of which don't run on Microsoft Windows and therefore, don't support any kind of standard framebuffer format. Because a programmer will likely be unfamiliar with this proprietary framebuffer layout, writing directly to it would produce unpredictable results. Furthermore, some OpenGL devices might not have a framebuffer that the CPU can address.

You can read the contents of the color, depth, and stencil buffers with the glReadPixels() command. Likewise, glDrawPixels() and glCopyPixels() are available for sending images to and BLTing images around in the OpenGL buffers.

14.015 How do I use glDrawPixels() and glReadPixels()?

glDrawPixels() and glReadPixels() write and read rectangular areas to and from the framebuffer, respectively. Also, you can access stencil and depth buffer information with the *format* parameter. Single pixels can be written or read by specifying *width* and *height* parameters of 1.

glDrawPixels() draws pixel data with the current raster position at the lower left corner. Problems using glDrawPixels() typically occur because the raster position is set incorrectly. When the raster position is set with the glRasterPos*() function, it is transformed as if it were a 3D vertex. Then the glDrawPixels() data is written to the resulting device coordinate raster position. (This allows you to tie pixel arrays and bitmap data to positions in 3D space).

When the raster position is outside the view volume, it's clipped and the glDrawPixels() call isn't rendered. This occurs even when part of the glDrawPixels() data would be visible. Here's info on how to render when the raster position is clipped.

glReadPixels() doesn't use the raster position. Instead, it obtains its (X,Y) device coordinate address from its first two parameters. Like glDrawPixels(), the area read has *x* and *y* for the lower left corner. Problems can occur when reading pixels if:

- The area being read is from a window that is overlapped or partially offscreen. glReadPixels() will return undefined data for the obscured area. (More info.)
- Memory wasn't allocated for the return data (the 7th parameter is a NULL pointer) causing a segmentation fault, core dump, or program termination. If you think you've allocated enough memory, but you still run into this problem, try doubling the amount of memory you've allocated. If this causes your read to succeed, chances are you've miscalculated the amount of memory needed.

For both glDrawPixels() and glReadPixels(), keep in mind:

- The *width* and *height* parameters are in pixels.
- If the drawn or read pixel data seems correct, but is slightly off, make sure you've set alignment correctly. Argument values are controlled with the glPixelStore*() functions. The PACK and UNPACK values control sending and receiving pixel data, from and to OpenGL, respectively.

14.020 How do I change between double- and single-buffered mode, in an existing a window?

If you create a single–buffered window, you can't change it.

If you create a double–buffered window, you can treat it as a single–buffered window by setting glDrawBuffer() to GL_FRONT and replacing your swap buffers call with a glFlush() call. To switch back to double–buffered, you need to set glDrawBuffer() to GL_BACK, and call swap buffers at the end of the frame.

14.030 How do I read back a single pixel?

Use glReadPixels(), passing a value of one for the *width* and *height* parameters.

14.040 How do I obtain the Z value for a rendered primitive?

You can obtain a single pixel's depth value by reading it back from the depth buffer with a call to glReadPixels(). This returns the screen space depth value.

It could be useful to have this value in object coordinate space. If so, you'll need to pass the window X and Y values, along with the screen space depth value to gluUnProject(). <u>See more information on gluUnProject() here</u>.

14.050 How do I draw a pattern into the stencil buffer?

You can set up OpenGL state as follows:

glEnable(GL_STENCIL_TEST); glStencilFunc(GL_ALWAYS, 0x1, 0x1); glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);

Subsequent rendering will set a 1 bit in the stencil buffer for every pixel rendered.

14.060 How do I copy from the front buffer to the back buffer and vice versa?

You need to call glCopyPixels(). The source and destination of glCopyPixels() are set with calls to glReadBuffer() and glDrawBuffer(), respectively. Thus, to copy from the back buffer to the front buffer, you can code the following:

glReadBuffer (GL_BACK); glDrawBuffer (GL_FRONT); glCopyPixels (GL_COLOR);

14.070 Why don't I get valid pixel data for an overlapped area when I call glReadPixels() where part of the window is overlapped by another window?

This is due to a portion of the OpenGL specification called the Pixel Ownership test. If a

OpenGL FAQ and Troubleshooting Guide

window is obscured by another window, it doesn't have to store pixel data for the obscured region. Therefore, a glReadPixels() call can return undefined data for the obscured region.

The Pixel Ownership test varies from one OpenGL implementation to the next. Some OpenGL implementations store obscured regions of a window, or the entire window, in an off-screen buffer. Such an implementation can return valid pixel data for an obscured window. However, many OpenGL implementations map pixels on the screen one-to-one to framebuffer storage locations and don't store (and can't return) pixel data for obscured regions of a window.

One strategy is to instruct the windowing system to bring the window forward to the top of the window stack, render, then perform the glReadPixels() call. However, such an approach still risks user intervention that might obscure the source window.

An approach that might work for some applications is to render into a nonvisible window, such as a Pixmap under X Windows. This type of drawing surface can't be obscured by the user, and its contents should always pass the pixel ownership test. Reading from such a drawing surface should always yield valid pixel data. Unfortunately, rendering to such drawing surfaces is often not accelerated by graphics hardware.

14.080 Why does the appearance of my smooth-shaded quad change when I view it with different transformations?

An OpenGL implementation may or may not break up your quad into two triangles for rendering. Whether it breaks it up or not (and if it does, the method used to split the quad) will determine how color is interpolated along the edges and ultimately across each scanline.

Many OpenGL applications avoid quads altogether because of their inherent rasterization problems. A quad can be rendered easily as a two-triangle GL_TRIANGLE_STRIP primitive with the same data transmission cost as the equivalent quad. Wise programmers use this primitive in place of quads.

14.090 How do I obtain exact pixelization of lines?

The OpenGL specification allows for a wide range of line rendering hardware, so exact pixelization may not be possible at all.

You might want to read the OpenGL specification and become familiar yourself with the diamond exit rule. Being familiar with this rule will give you the best chance to obtain exact pixelization. Briefly, the diamond exit rule specifies that a diamond–shaped area exists within each pixel. A pixel is rasterized by a line only if the mathematical definition of that line exits the diamond inscribed within that pixel.

14.100 How do I turn on wide-line endpoint capping or mitering?

OpenGL draws wide lines by rendering multiple width–1 component lines adjacent to each other. If the wide line is Y major, the component lines are offset in X; if the wide line is X major, the component lines are offset in Y. This can produce ugly gaps at the junction of line segments and differences in apparent width depending on the line segment's slope.

OpenGL doesn't provide a mechanism to cleanly join lines that share common vertices nor to

cleanly cap the endpoints.

One possible solution is to render smooth (antialiased) lines instead of normal aliased lines. To produce a clean junction, you need to draw lines with depth test disabled or the depth function set to GL_ALWAYS. See the question on rendering antialiased lines for more info.

Another solution is for the application to handle the capping and mitering. Instead of rendering lines, the application needs to render face–on polygons. The application will need to perform the necessary math to calculate the vertex locations to provide the desired capping and joining styles.

14.110 How do I render rubber-band lines?

The unspoken objective of this question is, "How can I render something, then erase it without disturbing what has already been rendered?"

Here are two common approaches.

One way is to use overlay planes. You draw the rubber–band lines into the overlay planes, then clear the overlay planes. The contents of the main framebuffer isn't disturbed. The disadvantage of this approach is that OpenGL devices don't widely support overlay planes.

The other approach is to render with logic op enabled and set to XOR mode. Assuming you're rendering into an RGBA window, your code needs to look like:

glEnable(GL_COLOR_LOGIC_OP);
glLogicOp(GL_XOR);

Set the color to white and render your lines. Where your lines are drawn, the contents of the framebuffer will be inverted. When you render the lines a second time, the contents of the framebuffer will be restored.

The logic op command for RGBA windows is only available with OpenGL 1.1. Under 1.0, you can only enable logic op in color index windows, and GL_LOGIC_OP is passed as the parameter to glEnable().

14.120 If I draw a quad in fill mode and again in line mode, why don't the lines hit the same pixels as the filled quad?

Filled primitives and line primitives follow different rules for rasterization.

When a filled primitive is rendered, a pixel is only touched if its exact center falls within the primitive's mathematical boundary.

When a line primitive is rasterized, ideally a pixel is only touched if the line exits a diamond inscribed in the pixel's boundary.

From these rules, it should be clear that a line loop specified with the same vertices as those used for a filled primitive, can rasterize pixels that the filled primitive doesn't.

(The OpenGL specification allows for some deviation from the diamond exit line rasterization rule, but it makes no difference in this scenario.)

14.130 How do I draw a full-screen quad?

See this question in the Transformation section.

14.140 How do I initialize or clear a buffer without calling glClear()?

Draw a full screen quad. See the Transformation section.

14.150 How can I make line or polygon antialiasing work?

To render smooth (antialiased) lines, an application needs to do the following:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_LINE_SMOOTH);
```

If the scene consists entirely of smooth lines, you need to disable the depth test or set it to GL_ALWAYS.

If a scene contains both smooth lines and other primitives, turning depth test off isn't an option. You can achieve nearly correct rendering results if you treat the smooth lines as transparent primitives. The other (non–blended) primitives should be rendered first, then the smooth lines rendered last, in back to front order. <u>See the transparency section</u> for more information.

Even taking these precautions might not prevent some rasterization artifacts at the joints of smooth line segments that share common vertices. The fact that the depth test is enabled could conceivably cause some line endpoints to be rendered incorrectly. This is a rendering artifact that you may have to live with if the depth test must be enabled while smooth lines are rendered.

Not all OpenGL implementations support antialiased polygons. According to the OpenGL spec, an implementation can render an aliased polygon when GL_POLYGON_SMOOTH is enabled.

14.160 How do I achieve full-scene antialiasing?

See the <u>OpenGL Programming Guide</u>, <u>Third Edition</u>, p452, for a description of a multi–pass accumulation buffer technique. This method performs well on devices that support the accumulation buffer in hardware.

On OpenGL 1.2 implementations that support the optional imaging extension, a smoothing filter may be applied to the final framebuffer image.

Many devices support the multisampling extension.

15 Transparency, Translucency, and Blending

15.010 What is the difference between transparent, translucent, and blended primitives?

A transparent physical material shows objects behind it as unobscured and doesn't reflect light off its surface. Clear glass is a nearly transparent material. Although glass allows most light to pass through unobscured, in reality it also reflects some light. A perfectly transparent material is completely invisible.

A translucent physical material shows objects behind it, but those objects are obscured by the translucent material. In addition, a translucent material reflects some of the light that hits it, making the material visible. Physical examples of translucent materials include sheer cloth, thin plastic, and smoked glass.

Transparent and translucent are often used synonymously. Materials that are neither transparent nor translucent are opaque.

Blending is OpenGL's mechanism for combining color already in the framebuffer with the color of the incoming primitive. The result of this combination is then stored back in the framebuffer. Blending is frequently used to simulate translucent physical materials. One example is rendering the smoked glass windshield of a car. The driver and interior are still visible, but they are obscured by the dark color of the smoked glass.

15.020 How can I achieve a transparent effect?

OpenGL doesn't support a direct interface for rendering translucent (partially opaque) primitives. However, you can create a transparency effect with the blend feature and carefully ordering your primitive data. You might also consider using <u>screen door</u> transparency.

An OpenGL application typically enables blending as follows:

glEnable (GL_BLEND);
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

After blending is enabled, as shown above, the incoming primitive color is blended with the color already stored in the framebuffer. glBlendFunc() controls how this blending occurs. The typical use described above modifies the incoming color by its associated alpha value and modifies the destination color by one minus the incoming alpha value. The sum of these two colors is then written back into the framebuffer.

The primitive's opacity is specified using glColor4*(). RGB specifies the color, and the alpha parameter specifies the opacity.

When using depth buffering in an application, you need to be careful about the order in which you render primitives. Fully opaque primitives need to be rendered first, followed by partially opaque primitives in back–to–front order. If you don't render primitives in this order, the primitives, which would otherwise be visible through a partially opaque primitive, might lose the depth test entirely.

15.030 How can I create screen door transparency?

This is accomplished by specifying a polygon stipple pattern with glPolygonStipple() and by

15 Transparency, Translucency, and Blending

rendering the transparent primitive with polygon stippling enabled (glEnable(GL_POLYGON_STIPPLE)). The number of bits set in the stipple pattern determine the amount of translucency and opacity; setting more bits result in a more opaque object, and setting fewer bits results in a more translucent object. Screendoor transparency is sometimes preferable to blending, becuase it's order independent (primitives don't need to be rendered in back-to-front order).

15.040 How can I render glass with OpenGL?

This question is difficult to answer, because what looks like glass to one person might not to another. What follows is a general algorithm to get you started.

First render all opaque objects in your scene. Disable lighting, enable blending, and render your glass geometry with a small alpha value. This should result in a faint rendering of your object in the framebuffer. (Note: You may need to sort your glass geometry, so it's rendered in back to front Z order.)

Now, you need to add the specular highlight. Set your ambient and diffuse material colors to black, and your specular material and light colors to white. Enable lighting. Set glDepthFunc(GL_EQUAL), then render your glass object a second time.

15.050 Do I need to render my primitives from back to front for correct rendering of translucent primitives to occur?

If your hardware supports destination alpha, you can experiment with different glBlendFunc() settings that use destination alpha. However, this won't solve all the problems with depth buffered translucent surfaces. The only sure way to achieve visually correct results is to sort and render your primitives from back to front.

15.060 I want to use blending but can't get destination alpha to work. Can I blend or create a transparency effect without destination alpha?

Many OpenGL devices don't support destination alpha. In particular, the OpenGL 1.1 software rendering libraries from Microsoft don't support it. The OpenGL specification doesn't require it.

If you have a system that supports destination alpha, using it is a simple matter of asking for it when you create your window. For example, pass GLUT_ALPHA to glutInitDisplayMode(), then set up a blending function that uses destination alpha, such as:

glBlendFunc(GL_ONE_MINUS_DST_ALPHA,GL_DST_ALPHA);

Often this question is asked under the mistaken assumption that destination alpha is required to do blending. It's not. You can use blending in many ways to obtain a transparency effect that uses source alpha instead of destination alpha. The fact that you might be on a platform without destination alpha shouldn't prevent you from obtaining a transparency effect. See the *OpenGL Programming Guide* chapter 6 for ways to use blending to achieve transparency.

15.070 If I draw a translucent primitive and draw another primitive behind it, I expect the second primitive to show through the first, but it's not there?

Is depth buffering enabled?

If you're drawing a polygon that's behind another polygon, and depth test is enabled, then the new polygon will typically lose the depth test, and no blending will occur. On the other hand, if you've disabled depth test, the new polygon will be blended with the existing polygon, regardless of whether it's behind or in front of it.

15.080 How can I make part of my texture maps transparent or translucent?

It depends on the effect you're trying to achieve.

If you want blending to occur after the texture has been applied, then use the OpenGL blending feature. Try this:

glEnable (GL_BLEND);
glBlendFunc (GL_ONE, GL_ONE);

You might want to use the alpha values that result from texture mapping in the blend function. If so, (GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA) is always a good function to start with.

However, if you want blending to occur when the primitive is texture mapped (i.e., you want parts of the texture map to allow the underlying color of the primitive to show through), then don't use OpenGL blending. Instead, you'd use glTexEnv(), and set the texture environment mode to GL_BLEND. In this case, you'd want to leave the texture environment color to its default value of (0,0,0,0).

16 Display Lists and Vertex Arrays

16.010 Why does a display list take up so much memory?

An OpenGL display list must make a copy of all data it requires to recreate the call sequence that created it. This means that for every glVertex3f() call, for example, the display list must provide storage for 3 values (usually 32–bit float values in most implementations). This is where most of the memory used by a typical display list goes.

However, in most implementations, there's also some memory that's needed to manage the display lists of a given context and other overhead. In certain pathological cases, this overhead memory can be larger than the memory used to store the display list data!

16.020 How can I share display lists between different contexts?

If you're using Microsoft Windows, use the wglShareLists() function. If you are using GLX, see the *share* parameter to glXCreateContext().

GLUT does not allow display list sharing. You can obtain the GLUT source, and make your own glutCreateWindow() and glutSetWindow() function calls. You can then modify the source to expose display list sharing. When doing so, you need to make sure your modified routines still work with the rest of GLUT.

16.030 How does display list nesting work? Is the called list copied into the calling list?

No. Only the call to the enclosed display list is copied into the parent list. This way a program can delete or replace a child list, call the parent, and see changes that were made.

16.040 Can I do a particular function while a display list is called?

A display list call is an atomic operation and therefore, it can't be interrupted. You can't call part of it, for example, then do something, then call the rest of it. Nor can you have a display list somehow signal your program from some point within the list.

However, an application doesn't have to create one large monolithic display list. By creating several smaller lists to call sequentially, an application is free to perform tasks between calls to glCallList().

An application can also use multithreading, so one thread can perform one task while another thread is calling a display list.

16.050 How can I change an OpenGL function call in a display list that contains many other OpenGL function calls?

OpenGL display lists aren't editable, so you can't modify the call sequence in them or even see which calls are embedded in them.

One way of creating a pseudo–editable display list is to create a hierarchical display list. (i.e., create a display list parent that contains calls to glCallList()). Then you can edit the display list by replacing the child display lists that the parent list references.

16.060 How can I obtain a list of function calls and OpenGL call parameters from a display list?

Currently, there isn't a way to programatically obtain either the function calls contained within a list or the parameters to those calls. An application that requires this information must track the data stored in a display list.

One option is to use an <u>OpenGL call logging utility</u>. These utilities capture the OpenGL calls a program makes, enabling you to see the calls that an application stores in a display list.

16.070 I've converted my program to use display lists, and it doesn't run any faster. Why not?

Achieving the highest performance from display lists is highly dependent on the OpenGL implementation, but here are a few pointers:

First, make sure that your application's process size isn't becoming so large that it's causing memory thrashing. Using display lists generally takes more memory than immediate mode, so it's possible that your program is spending more time thrashing memory blocks than rendering OpenGL calls.

Display lists won't improve the performance of a fill–limited application. Try rendering to a smaller window, and if your application runs faster, it's likely that it's fill–limited.

Stay away from GL_COMPILE_AND_EXECUTE mode. Instead, create the list using GL_COMPILE mode, then execute it with glCallList().

In some cases if you group your state changes together, the display list can optimize them as a group (i.e., it can remove redundant state changes, concatenate adjacent matrix changes, etc.).

Read the section on Performance for other tips.

16.080 To save space, should I convert all my coordinates to short before storing them in a display list?

No. Most implementations will convert your data to an internal format for storage in the display list anyway, and usually, that format will be single-precision float.

16.090 Will putting textures in a display list make them run faster?

In some implementations, a display list can optimize texture download and use of texture memory. In OpenGL 1.0, storing texture maps in display lists was the preferred method for optimizing texture performance. However, it resulted in increased memory usage in many implementations. Many vendors rallied around a better solution, <u>texture objects</u>, introduced in OpenGL 1.1. If your app is running on OpenGL 1.1 or later, texture objects are preferred.

16.100 Will putting vertex arrays in a display list make them run faster?

It depends on the implementation. In most implementations, it might decrease performance because of the increased memory use. However, some implementations may cache display lists on the graphics hardware, so the benefits of this caching could easily offset the extra memory usage.

16.110 When sharing display lists between contexts, what happens when I delete a display list in one context? Do I have to delete it in all the contexts to make it really go away?

When a display list is modified in one context (deleting is a form of modification), the results of that modification are immediately available in all shared contexts. So, deleting a display list in one context will cause it to cease to exist in all contexts in which it was previously visible.

16.120 How many display lists can I create?

There isn't a limit based on the OpenGL spec. Because a display list ID is a GLuint, 2^{32} display list identifiers are available. A more practical limit to go by is system memory resources.

16.130 How much memory does a display list use?

See the first question in this section. It depends on the implementation.

16.140 How will I know if the memory used by a display list has been freed?

This depends on the implementation. Some implementations free memory as soon as a display list is deleted. Others won't free the memory until it's needed by another display list or until the process dies.

16.150 How can I use vertex arrays to share vertices?

Because vertex arrays let you access a set of vertices and data by index, you might believe that they're designed to optimally share vertices. Indeed, a programmer new to vertex arrays might try to render a cube, in which each vertex is shared by three faces. The futility of this becomes obvious when you add normals for lighting and each instance of the shared vertex requires a unique normal. The only way to render a cube with normals is to include multiple copies of each vertex.

Vertex arrays weren't designed to improve vertex sharing. They were intended to let the programmer to specify blocks of dynamic geometry data with as few function calls as possible.

You can share vertices with vertex arrays the same way you do with OpenGL immediate mode, by the type of primitive used. GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, and GL_QUAD_STRIP share vertices between their component line segments, triangles, and quads. Other primitives do not. The type of primitive you choose to use when using vertex arrays determines whether you share vertices.

Note, however, that sharing vertices is implementation dependent. The OpenGL Specification dictates vertex array behavior, and as long as an OpenGL implementation conforms to spec, it's free to optimize vertex sharing in vertex arrays.

Some implementations feature the EXT_compiled_vertex_array extension, which is explicitly designed to let implementations share transformed vertex array data.

17 Using Fonts

17.010 How can I add fonts to my OpenGL scene?

OpenGL doesn't provide direct font support, so the application must use any of OpenGL's other features for font rendering, such as drawing bitmaps or pixmaps, creating texture maps containing an entire character set, drawing character outlines, or creating 3D geometry for each character.

Use bitmaps or pixmaps

The most straightforward method for rendering simple fonts is to use a glBitmap() or glDrawPixels() call for each character. The result is simple 2D text, which is suitable for labeling GUI controls, annotating 3D parts, etc.

glBitmap() is the fastest and simplest of the two, and renders characters in the current color. You can also use glDrawPixels() if required. However, note that glDrawPixels() always draws a rectangle, so if you desire a transparent background, it must be removed with alpha test and/or blending.

Typically, each glBitmap() call, one for every glyph in the font, is stored in an individual display list, which is indexed by its ASCII character value. Thus, a single call to glCallLists() can render an entire string of characters.

In X Windows, the glXUseXFont() call is available to create these display lists painlessly from a given font.

If you're using Microsoft Windows, look at the MSDN documentation for wglUseFontBitmaps(). It's conceptually identical to glXUseXFonts().

For GLUT, you need to use the glutBitmapCharacter() routine, which generates a bitmap for the specified character from the specified GLUT bitmap font.

Use texture mapping

In many OpenGL implementations, rendering glBitmap() and glDrawPixels() primitives is inherently slower than rendering an equivalent texture mapped quad. Use texture mapped primitives to render fonts on such devices.

The basic idea is to create a single texture map that contains all characters in a font (or at least all the characters that need to be rendered). To render an individual character, draw a texture mapped quad with texture coordinates configured to select the desired individual character. If desired, you can use alpha test to discard background pixels.

A library for using texture mapped fonts can be found here. It comes with source code.

Additional extensive information on texture mapped text and example code, can be found <u>here</u>.

The NeHe web page has a tutorial on using texture mapped fonts.

Stroked fonts

If you're using Microsoft Windows, look up the MSDN documentation on wglUseFontOutlines(). It contains example code for rendering stroked characters.

The glutStrokeCharacter() routine renders a single stroked character from a specified GLUT stroke font.

Geometric fonts

<u>The NeHe web page</u> has a tutorial for rendering geometric fonts. Look for the tutorial on outline fonts.

17.020 How can I use TrueType fonts in my OpenGL scene?

<u>The NeHe web page</u> has tutorials that show how to use TrueType fonts in a variety of ways.

See the Free Type library.

17.030 How can I make 3D letters, which I can light, shade, and rotate?

See <u>the NeHe web page</u> for a tutorial on using geometric fonts. Look for the tutorial on outline fonts.

See the Free Type library.

GLTT supports geometric TrueType fonts in OpenGL. It was formerly available from http://www.moonlight3d.org/gltt/, but fortunately is still available around the Web. Download GLTT v 2.4 (~125KB).

Glut 3.7 has an example called progs/contrib/text3d.c that may be informative.

18 Lights and Shadows

18.010 What should I know about lighting in general?

You must specify normals along with your geometry, or you must generate them automatically with evaluators, in order for lighting to work as expected. This is covered in question <u>18.020</u>.

Lighting does not work with the current color as set by glColor*(). It works with material colors. Set the material colors with glMaterial*(). Material colors can be made to track the current color with the color material feature. To use color material, call glEnable(GL_COLOR_MATERIAL). By default, this causes ambient and diffuse material colors to track the current color. You can specify which material color tracks the current color with a call to glColorMaterial().

Changing the material colors with color material and glColor*() calls may be more efficient than using glMaterial*(). See <u>question 18.080</u> for more information.

Lighting is computed at each vertex (and interpolated across the primitive, when glShadeModel() is set to GL_SMOOTH). This may cause primitives to appear too dark, even though a light is centered over the primitive. You can obtain more correct lighting with a higher surface approximation, or by using <u>light maps</u>.

A light's position is transformed by the current ModelView matrix at the time the position is specified with a call to glLight*(). This is analogous to how geometric vertices are transformed by the current ModelView matrix when they are specified with a call to glVertex*(). For more information on positioning your light source, see <u>question 18.050</u>.

18.020 Why are my objects all one flat color and not shaded and illuminated?

This effect occurs when you fail to supply a normal at each vertex.

OpenGL needs normals to calculate lighting equations, and it won't calculate normals for you (with the exception of evaluators). If your application doesn't call glNormal*(), then it uses the default normal of (0.0, 0.0, 1.0) at every vertex. OpenGL will then compute the same, or nearly the same, lighting result at each vertex. This will cause your model to look flat and lack shading.

The solution is to simply calculate the normals that need to be specified at any given vertex. Then send them to OpenGL with a call to glNormal3f() just prior to specifying the vertex, which the normal is associated with.

If you don't know how to calculate a normal, in most cases you can do it simply with a vector cross product. The <u>OpenGL Programming Guide</u> contains a small section explaining how to calculate normals. Also most basic 3D computer graphics books cover it, because it's not OpenGL–specific.

18.030 How can I make OpenGL automatically calculate surface normals?

OpenGL won't do this unless you're using evaluators.

18.040 Why do I get only flat shading when I light my model?

First, check the obvious. glShadeModel() should be set to GL_SMOOTH, which is the default value, so if you haven't called glShadeModel() at all, it's probably already set to GL_SMOOTH, and something else is wrong.

If glShadeModel() is set correctly, the problem is probably with your surface normals. To achieve a smooth shading effect, generally you need to specify a different normal at each vertex. If you have set the same normal at each vertex, the result, in most cases, will be a flatly shaded primitive.

Keep in mind that a typical surface normal is perpendicular to the surface that you're attempting to approximate.

This scenario can be tough to debug, especially for large models. The best debugging approach is to write a small test program that draws only one primitive, and try to reproduce the problem. It's usually easy to use a debugger to isolate and fix a small program, which reproduces the problem.

18.050 How can I make my light move or not move and control the light position?

First, you must understand how the light position is transformed by OpenGL.

The light position is transformed by the contents of the current top of the ModelView matrix stack when you specify the light position with a call to glLightfv(GL_LIGHT_POSITION,...). If you later change the ModelView matrix, such as when the view changes for the next frame, the light position isn't automatically retransformed by the new contents of the ModelView matrix. If you want to update the light's position, you must again specify the light position with a call to glLightfv(GL_LIGHT_POSITION,...).

Asking the question "how do I make my light move" or "how do I make my light stay still" usually doesn't provide enough information to answer the question. For a better answer, you need to be more specific. Here are some more specific questions, and their answers:

• How can I make my light position stay fixed relative to my eye position? How do I make a headlight?

You need to specify your light in eye coordinate space. To do so, set the ModelView matrix to the identity, then specify your light position. To make a headlight (a light that appears to be positioned at or near the eye and shining along the line of sight), set the ModelView to the identity, set the light position at (or near) the origin, and set the direction to the negative Z axis.

When a light's position is fixed relative to the eye, you don't need to respecify the light position for every frame. Typically, you specify it once when your program initializes.

• How can I make my light stay fixed relative to my scene? How can I put a light in the corner and make it stay there while I change my view?

As your view changes, your ModelView matrix also changes. This means you'll need to respecify the light position, usually at the start of every frame. A typical application will

display a frame with the following pseudocode:

Set the view transform. Set the light position //glLightfv(GL_LIGHT_POSITION,133;) Send down the scene or model geometry. Swap buffers.

If your light source is part of a light fixture, you also may need to specify a modeling transform, so the light position is in the same location as the surrounding fixture geometry.

• How can I make a light that moves around in a scene?

Again, you'll need to respecify this light position every time the view changes. Additionally, this light has a dynamic modeling transform that also needs to be in the ModelView matrix before you specify the light position. In pseudocode, you need to do something like:

Set the view transform Push the matrix stack Set the model transform to update the light146;s position Set the light position //glLightfv(GL_LIGHT_POSITION,133;) Pop the matrix stack Send down the scene or model geometry Swap buffers.

18.060 How can I make a spotlight work?

A spotlight is simply a point light source with a small light cone radius. Alternatively, a point light is just a spot light with a 180 degree radius light cone. Set the radius of the light cone by changing the cutoff parameter of the light:

glLightf (GL_LIGHT1, GL_SPOT_CUTOFF, 15.f);

The above call sets the light cone radius to 15 degrees for light 1. The light cone's total spread will be 30 degrees.

A spotlight's position and direction are set as for any normal light.

18.070 How can I create more lights than GL_MAX_LIGHTS?

First, make sure you really need more than OpenGL provides. For example, when rendering a street scene at night with many buildings and streetlights, you need to ask yourself: Does every building need to be illuminated by every single streetlight? When light attenuation and direction are accounted for, you may find that any given piece of geometry in your scene is only illuminated by a small handful of lights.

If this is the case, you need to reuse or cycle the available OpenGL lights as you render your scene.

The GLUT distribution comes with a small example that might be informative to you. It's called multilight.c.

If you really need to have a single piece of geometry lit by more lights than OpenGL provides, you'll need to simulate the effect somehow. One way is to calculate the lighting for

some or all the lights. Another method is to use texture maps to simulate lighting effects.

18.080 Which is faster: making glMaterial*() calls or using glColorMaterial()?

Within a glBegin()/glEnd() pair, on most OpenGL implementations, a call to glColor3f() generally is faster than a call to glMaterialfv(). This is simply because most implementations tune glColor3f(), and processing a material change can be complex and difficult to optimize. For this reason, glColorMaterial() generally is recognized as the most efficient way to change an object's material color.

18.090 Why is the lighting incorrect after I scale my scene to change its size?

The OpenGL specification needs normals to be unit length to achieve typical lighting results. The current ModelView matrix transforms normals. If that matrix contains a scale transformation, transformed normals might not be unit length, resulting in undesirable lighting problems.

OpenGL 1.1 lets you call glEnable(GL_NORMALIZE), which will make all normals unit length after they're transformed. This is often implemented with a square root and can be expensive for geometry limited applications.

Another solution, available in OpenGL 1.2 (and as an extension to many 1.1 implementations), is glEnable(GL_RESCALE_NORMAL). Rather than making normals unit length by computing a square root, GL_RESCALE_NORMAL multiplies the transformed normal by a scale factor. If the original normals are unit length, and the ModelView matrix contains uniform scaling, this multiplication will restore the normals to unit length.

If the ModelView matrix contains nonuniform scaling, GL_NORMALIZE is the preferred solution.

18.100 After I turn on lighting, everything is lit. How can I light only some of the objects?

Remember that OpenGL is a state machine. You'll need to do something like this:

glEnable(GL_LIGHTING);
// Render lit geometry.
glDisable(GL_LIGHTING);
// Render non-lit geometry.

18.110 How can I use light maps (e.g., Quake-style) in OpenGL?

See this question in the Texture Mapping section.

18.120 How can I achieve a refraction lighting effect?

First, consider whether OpenGL is the right API for you. You might need to use a ray tracer to achieve complex light affects such as refraction.

If you're certain that you want to use OpenGL, you need to keep in mind that OpenGL doesn't provide functionality to produce a refraction effect. You'll need to fake it. The most likely solution is to calculate an image corresponding to the refracted rendering, and texture map it onto the surface of the primitive that's refracting the light.

18.130 How can I render caustics?

OpenGL can't help you render caustics, except for texture mapping. GLUT 3.7 comes with some demos that show you how to achieve caustic lighting effects.

18.140 How can I add shadows to my scene?

OpenGL does not support shadow rendering directly. However, any standard algorithm for rendering shadows can be used in OpenGL. Some algorithms are described at <u>http://www.opengl.org</u>. Follow the Coding Tutorials & Techniques link, then the Rendering Techniques link. Scroll down to the Lighting, Shadows, & Reflections section.

The GLUT 3.7 distribution comes with examples that demonstrate how to do this using projection shadows and the stencil buffer.

Projection shadows are ideal if your shadow is only to lie on a planar object. You can generate geometry of the shadow using glFrustum() to transform the object onto the projection plane.

Stencil buffer shadowing is more flexible, allowing shadows to lie on any object, planar or otherwise. The basic algorithm is to calculate a "shadow volume". Cull the back faces of the shadow volume and render the front faces into the stencil buffer, inverting the stencil values. Then render the shadow volume a second time, culling front faces and rendering the back faces into the stencil buffer, again inverting the stencil value. The result is that the stencil planes will now contain non-zero values where the shadow should be rendered. Render the scene a second time with only ambient light enabled and glDepthFunc() set to GL_EQUAL. The result is a rendered shadow.

Another mechanism for rendering shadows is outlined in the SIGGRAPH '92 paper *Fast Shadows and Lighting Effects Using Texture Mapping*, Mark Segal et al. This paper describes a relatively simple extension to OpenGL for using the depth buffer as a shadow texture map. Both the GL_EXT_depth_texture and the GL_EXT_texture3D (or OpenGL 1.2) extensions are required to use this method.

19 Curves, Surfaces, and Using Evaluators

19.010 How can I use OpenGL evaluators to create a B-spline surface?

OpenGL evaluators use a Bezier basis. To render a surface using any other basis, such as B–spline, you must convert your control points to a Bezier basis. The <u>OpenGL Programming</u><u>Guide</u>, Chapter 12, lists a number of reference books that cover the math behind these conversions.

19.020 How can I retrieve the geometry values produced by evaluators?

OpenGL does not provide a straightforward mechanism for this.

You might <u>download the Mesa source code distribution</u>, and modify its evaluator code to return object coordinates rather than pass them into the OpenGL geometry pipeline.

Evaluators involve a lot of math, so their performance in immediate mode is sometimes unacceptable. Some programmers think they need to "capture" the generated geometry, and play it back to achieve maximum performance. Indeed, this would be a good solution if it were possible. Some implementations provide maximum evaluator performance through the use of display lists.

20 Picking and Using Selection

20.010 How can I know which primitive a user has selected with the mouse?

OpenGL provides the <u>GL_SELECTION render mode</u> for this purpose. However, you can use other methods.

You might render each primitive in a unique color, then use glReadPixels() to read the single pixel under the current mouse location. Examining the color determines the primitive that the user selected. <u>Here's information on rendering each primitive in a unique color</u> and <u>information on using glDrawPixels()</u>.

Yet another method involves shooting a pick ray through the mouse location and testing for intersections with the currently displayed objects. OpenGL doesn't test for ray intersections (for how to do, see the BSP FAQ), but you'll need to interact with OpenGL to generate the pick ray.

One way to generate a pick ray is to call <u>gluUnProject()</u> twice for the mouse location, first with *winz* of 0.0 (at the near plane), then with *winz* of 1.0 (at the far plane). Subtract the near plane call's results from the far plane call's results to obtain the XYZ direction vector of your ray. The ray origin is the view location, of course.

Another method is to generate the ray in eye coordinates, and transform it by the inverse of the ModelView matrix. In eye coordinates, the pick ray origin is simply (0, 0, 0). You can build the pick ray vector from the perspective projection parameters, for example, by setting up your perspective projection this way

```
aspect = double(window_width)/double(window_height);
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
glFrustum(-near_height * aspect,
    near_height * aspect,
    -near_height,
    near_height, zNear, zFar );
```

you can build your pick ray vector like this:

```
int window_y = (window_height - mouse_y) - window_height/2;
double norm_y = double(window_y)/double(window_height/2);
int window_x = mouse_x - window_width/2;
double norm_x = double(window_x)/double(window_width/2);
```

(Note that most window systems place the mouse coordinate origin in the upper left of the window instead of the lower left. That's why *window_y* is calculated the way it is in the above code. When using a glViewport() that doesn't match the window height, the viewport height and viewport Y are used to determine the values for *window_y* and *norm_y*.)

The variables *norm_x* and *norm_y* are scaled between -1.0 and 1.0. Use them to find the mouse location on your *zNear* clipping plane like so:

float y = near_height * norm_y;
float x = near_height * aspect * norm_x;

Now your pick ray vector is (x, y, -zNear).

To transform this eye coordinate pick ray into object coordinates, multiply it by the inverse of the ModelView matrix in use when the scene was rendered. When performing this multiplication, remember that the pick ray is made up of a vector and a point, and that vectors and points transform differently. You can translate and rotate points, but vectors only rotate. The way to guarantee that this is working correctly is to define your point and vector as four–element arrays, as the following pseudo–code shows:

float ray_pnt[4] = {0.f, 0.f, 0.f, 1.f};
float ray_vec[4] = {x, y, -near_distance, 0.f};

The one and zero in the last element determines whether an array transforms as a point or a vector when multiplied by the inverse of the ModelView matrix.

20.020 What do I need to know to use selection?

Specify a selection buffer:

```
GLuint buffer[BUF_SIZE];
glSelectBuffer (BUF_SIZE, buffer);
```

Enter selection mode, render as usual, then exit selection mode:

GLint hits; glRenderMode(GL_SELECT); // ...render as usual... hits = glRenderMode(GL_RENDER);

The call to glRenderMode(GL_RENDER) exits selection mode and returns the number of hit records stored in the selection buffer. Each hit record contains information on the primitives that were inside the view volume (controlled with the ModelView and Projection matrices).

That's the basic concept. In practice, you may want to restrict the view volume. The gluPickMatrix() function is a handy method for restricting the view volume size to within a set number of pixels away from a given (X,Y) position, such as the current mouse or cursor location.

You'll also want to use the name stack to specify unique names for primitives of interest. After the stack is pushed once, any number of different names may be loaded onto the stack. Typically, load a name, then render a primitive or group of primitives. The name stack allows for selection to occur on heirarchical databases.

After returning to GL_RENDER render mode, you'll need to parse the selection buffer. It will contain zero or more hit records. The number of hit records is returned by the call to glRenderMode(GL_RENDER). Each hit record contains the following information stored as unsigned ints:

- Number of names in the name stack for this hit record
- Minimum depth value of primitives (range 0 to 2^{32} -1)
- Maximum depth value of primitives (range 0 to $2^{32}-1$)
- Name stack contents (one name for each unsigned int).

You can use the minimum and maximum Z values with the device coordinate X and Y if known (perhaps from a mouse click) to determine an object coordinate location of the picked primitive. You can scale the Z

values to the range 0.0 to 1.0, for example, and use them in a call to gluUnProject().

20.030 Why doesn't selection work?

This is usually caused by one of two things.

Did you account for the inverted Y coordinate? Most window systems (Microsoft Windows, X Windows, others?) usually return mouse coordinates to your program with Y=0 at the top of the window, while OpenGL assumes Y=0 is at the bottom of the window. Assuming you're using a default viewport, transform the Y value from window system coordinates to OpenGL coordinates as (windowHeight-y).

Did you set up the transformations correctly? Assuming you're using gluPickMatrix(), it should be loaded onto the Projection matrix immediately after a call to glLoadIdentity() and before you multiply your projection transform (using glFrustum(), gluPerspective(), glOrtho(), etc.). Your ModelView transformation should be the same as if you were rendering normally.

20.040 How can I debug my picking code?

A good technique for debugging picking or selection code is not to call glRenderMode(GL_SELECT). Simply comment out this function call in your code. The result is instead of performing a selection, your code will render the contents of the pick box to your window. This allows you to see visually what is inside your pick box.

Along with this method, it's generally a good idea to enlarge your pick box, so you can see more in your window.

20.050 How can I perform pick highlighting the way PHIGS and PEX provided?

There's no elegant way to do this, and that's why many former PHIGS and PEX implementers are now happy as OpenGL implementers. OpenGL leaves this up to the application.

After you've identified the primitive you need to highlight with selection, how you highlight it is up to your application. You might render the primitive into the displayed image in the front buffer with a different color set. You may need to use polygon offset to make this work, or at least set glDepthFunc(GL_EQUAL). You might only render the outline or render the primitive consecutive times in different colors to create a flashing effect.

21 Texture Mapping

21.010 What are the basic steps for performing texture mapping?

At the bare minimum, a texture map must be specified, texture mapping must be enabled, and appropriate texture coordinates must be set at each vertex. While these steps will produce a texture mapped primitive, typically they don't meet the requirements of most OpenGL 1.2 applications. Use the following steps instead.

- Create a texture object for each texture in use. The texture object stores the texture map and associated texture parameter state. See <u>question 21.070</u> for more information on texture objects.
- Store each texture map or mipmap pyramid in its texture object, along with parameters to control its use.
- On systems with limited texure memory, set the priority of each texture object with glPrioritizeTextures() to minimize texture memory thrashing.
- Whem your application renders the scene, bind each texture object before rendering the geomtry to be texture mapped. Enable and disable texture mapping as needed.

21.020 I'm trying to use texture mapping, but it doesn't work. What's wrong?

Check for the following:

- Texture mapping should be enabled, and a texture map must be bound (when using texture objects) or otherwise submitted to OpenGL (for example, with a call to glTexImage2D()).
- Make sure you understand the different wrap, environment, and filter modes that are available. Make sure you have set appropriate values.
- ♦ Keep in mind that texture objects don't store some texture parameters. Texture objects bind to a target (either GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D), and the texture object stores changes to those targets. glTexGen(), for example, doesn't change the state of the texture target, and therefore isn't part of texture objects.
- ♦ If you're using a mipmapping filter (e.g., you've called glTexParameter*(), setting a min or mag filter that has MIPMAP in its name), make sure you've set all levels of the mipmap pyramid. All levels must be set, or texture mapping won't occur. You can set all levels at the same time with the gluBuild2DMipmaps() function. All levels of the mipmap pyramid must have the same number of components.
- Remember that OpenGL is a state machine. If you don't specify texture coordinates, either explicitly with glTexCoord*(), or generated automatically with glTexGen()), then OpenGL uses the current texture coordinate for all vertices. This may cause some primitives to be texture mapped with a single color or single texel value.
- If you're using multiple rendering contexts and need to share texture objects between contexts, you must explicitly enable texture object sharing. This is done with the wglShareLists() function in Microsoft Windows and glXCreateContext() under X Windows.
- Check for errors with glGetError().

21.030 Why doesn't lighting work when I turn on texture mapping?

There are many well-meaning texture map demos available on the Web that set the texture environment to GL_DECAL or GL_REPLACE. These environment modes effectively replace the primitive color with the texture color. Because lighting values are calculated before texture mapping (lighting is a per vertex operation, while texture mapping is a per fragment operation), the texture color replaces the colors calculated by lighting. The result is that lighting appears to stop working when texture mapping is enabled.

The default texture environment is GL_MODULATE, which multiplies the texture color by the primitive (or lighting) color. Most applications that use both OpenGL lighting and texture mapping use the GL_MODULATE texture environment.

Look for the following line in your code:

glTexEnv (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL); /* or GL_REPLACE

You should change GL_DECAL to GL_MODULATE, or simply delete the line entirely (since GL_MODULATE is the default).

21.040 Lighting and texture mapping work pretty well, but why don't I see specular highlighting?

Your geometry may have a nice white specular highlight when it's not texture mapped, but when you apply a non-white texture suddenly the highlight goes away even though the geometry is still lit. This is because GL_MODULATE multiplies the primitive's lighting color components with the texture color components. For example, assume a white specular highlight is being multiplied by a red texture map. The final color is then (1.0*1.0, 1.0*0.0, 1.0*0.0) or (1.0, 0.0, 0.0), which is red. The white specular highlight isn't visible.

OpenGL 1.2 solves this problem by applying specular highlights after texture mapping. This separate specular lighting mode is turned on by:

glLightModel (GL_LIGHT_MODEL_COLOR_CONTROL,GL_SEPARATE_SPECULAR_COLOR);

By default, it's set to GL_SINGLE_COLOR, which maintains backwards compatibility with OpenGL 1.1 and earlier.

If you're not using OpenGL 1.2, other solutions are available. Many vendors provide proprietary extensions for allowing you to apply the specular highlight after the texture map. <u>See this example code</u> for how to do this on HP systems. Many OpenGL vendors have settled on an the <u>EXT separate specular color extension</u>.

Another method works on any OpenGL implementation, because it only uses regular OpenGL 1.0 functionality and doesn't depend on extensions. You need to render your geometry in two passes: first with normal lighting and texture mapping enabled, then the second pass will render the specular highlight. <u>See this example code</u> for a demonstration of how to do it.

21.050 How can I automatically generate texture coordinates?

Use the glTexGen() function.

21.060 Should I store texture maps in display lists?

See this question in the display list section.

21.070 How do texture objects work?

Texture objects store texture maps and their associated texture parameter state. They allow switching between textures with a single call to glBindTexture().

Texture objects were introduced in OpenGL 1.1. Prior to that, an application changed textures by calling glTexImage*(), a rather expensive operation. Some OpenGL 1.0 implementations simulated texture object functionality for <u>texture maps that were stored in display lists</u>.

Like display lists, a texture object has a GLuint identifier (the *textureName* parameter to glBindTexture()). OpenGL supplies your application with texture object names when your application calls glGenTextures(). Also like display lists, texture objects can be <u>shared across</u> rendering contexts.

Unlike display lists, texture objects are mutable. When a texture object is bound, changes to texture object state are stored in the texture object, including changes to the texture map itself.

The following functions affect and store state in texture objects: glTexImage*(), glTexSubImage*(), glCopyTexImage*(), glCopyTexSubImage*(), glTexParameter*(), and glPrioritizeTextures(). Since the GLU routines for building mipmap pyramids ultimately call glTexImage*(), they also affect texture object state.Noticeably absent from this list are glTexEnv*() and glTexGen*(); they do not store state in texture objects.

Here is a summary of typical texture object usage:

- Get a *textureName* from glGenTextures(). You'll want one name for each of the texture objects you plan to use.
- Initially bind a texture object with glBindTexture(). Specify the texture map, and any texture parameters. Repeat this for all texture objects your application uses.
- Before rendering texture mapped geometry, call glBindTexture() with the desired *textureName*. OpenGL will use the texture map and texture parameter state stored in that object for rendering.

21.080 Can I share textures between different rendering contexts?

Yes, if you use texture objects. Texture objects can be shared the same way <u>display lists can</u>. If you're using Microsoft Windows, see the wglShareLists() function. For a GLX platform, see the *share* parameter to glXCreateContext().

21.090 How can I apply multiple textures to a surface?

Note that EXT_multitexture and SGIS_multitexture are both obsolete. The preferred multitexturing extension is ARB_multitexture.

The ARB_multitexture spec is included in the OpenGL 1.2.1 spec: <u>http://www.opengl.org/Documentation/Specs.html</u>.

An example is on Michael Gold's Web page.

A useful snippet is available at

<u>http://reality.sgi.com/blythe/sig99/advanced99/notes/node48.html</u>. It's part of a wider presentation entitled <u>Advanced Graphics Programming Techniques Using OpenGL</u>. It's a useful supplement to anyone starting OpenGL and 3D graphics in general.

21.100 How can I perform light mapping?

You can simulate lighting by creating a texture map that mimics the light pattern and by applying it as a texture to the lit surface. After you've created the light texture map, there's nothing special about how you apply it to the surface. It's just like any other texture map. For this reason, this question really isn't specific to OpenGL.

The GLUT 3.7 distribution contains an example that uses texture mapping to simulate lighting called progs/advanced97/lightmap.c.

21.110 How can I turn my files, such as GIF, JPG, BMP, etc. into a texture map?

OpenGL doesn't provide support for this. With whatever libraries or home-brewed code you desire to read in the file, then by using the glTexImage2D call, transform the pixel data into something acceptable, and use it like any other texture map.

Source code for doing this with <u>TGA files can be found here</u>.

See the Miscellaneous section for info on reading and writing 2D image files.

21.120 How can I render into a texture map?

With OpenGL 1.1, you can use the glCopyTexImage2D() or glCopyTexSubImage2D() functions to assist with this task. glCopyTexImage2D() takes the contents of the framebuffer and sets it as the current texture map, while glCopyTexSubImage2D() only replaces part of the current texture with the contents of the framebuffer. There's a GLUT 3.7 example called multispheremap.c that does this.

21.130 What's the maximum size texture map my device will render hardware accelerated?

A good OpenGL implementation will render with hardware acceleration whenever possible. However, the implementation is free to not render hardware accelerated. OpenGL doesn't provide a mechanism to ensure that an application is using hardware acceleration, nor to query that it's using hardware acceleration. With this information in mind, the following may still be useful:

You can obtain an estimate of the maximum texture size your implementation supports with the following call:

GLint texSize; glGetIntegerv(GL_MAX_TEXTURE_SIZE, &texSize);

If your texture isn't hardware accelerated, but still within the size restrictions returned by GL_MAX_TEXTURE_SIZE, it should still render correctly.

This is only an estimate, because the glGet*() function doesn't know what *format*, *internalformat*, *type*, and other parameters you'll be using for any given texture. OpenGL 1.1 and greater solves this problem by

allowing texture proxy.

Here's an example of using texture proxy:

Note the *pixels* parameter is NULL, because OpenGL doesn't load texel data when the *target* parameter is GL_PROXY_TEXTURE_2D. Instead, OpenGL merely considers whether it can accommodate a texture of the specified size and description. If the specified texture can't be accommodated, the width and height texture values will be set to zero. After making a texture proxy call, you'll want to query these values as follows:

21.140 How can I texture map a sphere, cylinder, or any other object with multiple facets?

Texture map these objects using fractional texture coordinates. Each facet of an approximated surface or object will only show one small part of the texture map. Fractional texture coordinates determine what part of the texture map is applied to which facet.

22 Performance

22.010 What do I need to know about performance?

First, read chapters 11 through 14 of the book <u>OpenGL on Silicon Graphics Systems</u>. Although some of the information is SGI machine specific, most of the information applies to OpenGL programming on any platform. It's invaluable reading for the performance–minded OpenGL programmer.

Consider a performance tuning analogy: A database application spends 5 percent of its time looking up records and 95 percent of its time transmitting data over a network. The database developer decides to tune the performance. He sits down and looks at the code for looking up records and sees that with a few simple changes he can reduce the time it'll take to look up records by more than 50 percent. He makes the changes, compiles the database, and runs it. To his dismay, there's little or no noticeable performance increase!

What happened? The developer didn't identify the bottleneck before he began tuning. The most important thing you can do when attempting to boost your OpenGL program's performance is to identify where the bottleneck is.

Graphics applications can be bound in several places. Generally speaking, bottlenecks fall into three broad categories: CPU limited, geometry limited, and fill limited.

CPU limited is a general term. Specifically, it means performance is limited by the speed of the CPU. Your application may also be bus limited, in which the bus bandwidth prevents better performance. Cache size and amount of RAM can also play a role in performance. For a true CPU–limited application, performance will increase with a faster CPU. Another way to increase performance is to reduce your application's demand on CPU resources.

A geometry limited application is bound by how fast the computer or graphics hardware can perform vertex computations, such as transformation, clipping, lighting, culling, vertex fog, and other OpenGL operations performed on a per vertex basis. For many very low–end graphics devices, this processing is performed in the CPU. In this case, the line between CPU limited and geometry limited becomes fuzzy. In general, CPU limited implies that the bottleneck is CPU processing unrelated to graphics.

In a fill–limited application, the rate you can render is limited by how fast your graphics hardware can fill pixels. To go faster, you'll need to find a way to either fill fewer pixels, or simplify how pixels are filled, so they can be filled at a faster rate.

It's usually quite simple to discern whether your application is fill limited. Shrink the window size, and see if rendering speeds up. If it does, you're fill limited.

If you're not fill limited, then you're either CPU limited or geometry limited. One way to test for a CPU limitation is to change your code, so it repeatedly renders a static, precalculated scene. If the performance is significantly faster, you're dealing with a CPU limitation. The part of your code that calculates the scene or does other application–specific processing is causing your performance hit. You need to focus on tuning this part of your code.

If it's not fill limited and not CPU limited, congratulations! It's geometry limited. The per

vertex features you've enabled or the shear volume of vertices you're rendering is causing your performance hit. You need to reduce the geometry processing either by reducing the number of vertices or reducing the calculations OpenGL must use to process each vertex.

22.020 How can I measure my application's performance?

To measure an application's performance, note the system time, do some rendering, then note the system time again. The difference between the two system times tells you how long the application took to render. Benchmarking graphics is no different from benchmarking any other operations in a computer system.

Many graphics programmers often want to measure frames per second (FPS). A simple method is to note the system time, render a frame, and note the system time again. FPS is then calculated as $(1.0 / \text{elapsed_time})$. You can obtain a more accurate measurement by timing multiple frames. For example if you render 10 frames, FPS would be $(10.0 / \text{elapsed_time})$.

To obtain primitives or triangles per second, add a counter to your code for incrementing as each primitive is submitted for rendering. This counter needs to be reset to zero when the system time is initially obtained. If you already have a complex application that is nearly complete, adding this benchmarking feature as an afterthought might be difficult. When you intend to measure primitives per second, it's best to design your application with benchmarking in mind.

Calculating pixels per second is a little tougher. The easiest way to calculate pixels per second is to write a small benchmark program that renders primitives of a known pixel size.

GLUT 3.7 comes with a benchmark called progs/bucciarelli/gltest that measures OpenGL rendering performance and is free to download. You can also visit the <u>Standard Performance</u> <u>Evaluation Corporation</u>, which has many benchmarks you can download free, as well as the latest performance results from several OpenGL hardware vendors.

22.030 Which primitive type is the fastest?

GL_TRIANGLE_STRIP is generally recognized as the most optimal OpenGL primitive type. Be aware that the primitive type might not make a difference unless you're geometry limited.

22.040 What's the cost of redundant calls?

While some OpenGL implementations make redundant calls as cheap as possible, making redundant calls generally is considered bad practice. Certainly you shouldn't count on redundant calls as being cheap. Good application developers avoid them when possible.

22.050 I have (n) lights on, and when I turned on (n+1), suddenly performance dramatically drops. What happened?

Your graphics device supports (n) lights in hardware, but because you turned on more lights than what's supported, you were kicked off the hardware and are now rendering in the software. The only solution to this problem, except to use less lights, is to buy better hardware.

22.060 I'm using (n) different texture maps and when I started using (n+1) instead, performance drastically drops. What happened?

Your graphics device has a limited amount of dedicated texture map memory. Your (n) textures fit well in the texture memory, but there wasn't room left for any more texture maps. When you started using (n+1) textures, suddenly the device couldn't store all the textures it needed for a frame, and it had to swap them in from the computer's system memory. The additional bus bandwidth required to download these textures in each frame killed your performance.

You might consider using smaller texture maps at the expense of image quality.

22.070 Why are glDrawPixels() and glReadPixels() so slow?

While performance of the OpenGL 2D path (as its called) is acceptable on many higher–end UNIX workstation–class devices, some implementations (especially low–end inexpensive consumer–level graphics cards) never have had good 2D path performance. One can only expect that corners were cut on these devices or in the device driver to bring their cost down and decrease their time to market. When this was written (early 2000), if you purchase a graphics device for under \$500, chances are the OpenGL 2D path performance will be unacceptably slow.

If your graphics system should have decent performance but doesn't, there are some steps you can take to boost the performance.

First, all glPixelTransfer() state should be set to their default values. Also, glPixelStore() should be set to its default value, with the exception of GL_PACK_ALIGNMENT and GL_UNPACK_ALIGNMENT (whichever is relevant), which should be set to 8. Your data pointer will need to be correspondingly double– word aligned.

Second, examine the parameters to glDrawPixels() or glReadPixels(). Do they correspond to the framebuffer layout? Think about how the framebuffer is configured for your application. For example, if you know you're rendering into a 24–bit framebuffer with eight bits of destination alpha, your type parameter should be GL_RGBA, and your format parameter should be GL_UNSIGNED_BYTE. If your type and format parameters don't correspond to the framebuffer configuration, it's likely you'll suffer a performance hit due to the per pixel processing that's required to translate your data between your parameter specification and the framebuffer format.

Finally, make sure you don't have unrealistic expectations. Know your system bus and memory bandwidth limitations.

22.080 Is it faster to use absolute coordinates or to use relative coordinates?

By using absolute (or "world") coordinates, your application doesn't have to change the ModelView matrix as often. By using relative (or "object") coordinates, you can cut down on data storage of redundant primitives or geometry.

A good analogy is an architectural software package that models a hotel. The hotel model has hundreds of thousands of rooms, most of which are identical. Certain features are identical in each room, and maybe each room has the same lamp or the same light switch or doorknob.

The application might choose to keep only one doorknob model and change the ModelView matrix as needed to render the doorknob for each hotel room door. The advantage of this method is that data storage is minimized. The disadvantage is that several calls are made to change the ModelView matrix, which can reduce performance. Alternatively, the application could instead choose to keep hundreds of copies of the doorknob in memory, each with its own set of absolute coordinates. These doorknobs all could be rendered with no change to the ModelView matrix. The advantage is the possibility of increased performance due to less matrix changes. The disadvantage is additional memory overhead. If memory overhead gets out of hand, paging can become an issue, which certainly will be a performance hit.

There is no clear answer to this question. It's model- and application-specific. You'll need to benchmark to determine which method is best for your model or application.

22.090 Are display lists or vertex arrays faster?

Which is faster varies from system to system.

If your application isn't geometry limited, you might not see a performance difference at all between display lists, vertex arrays, or even immediate mode.

22.100 How do I make triangle strips out of triangles?

As mentioned in <u>22.030</u>, GL_TRIANGLE_STRIP is generally recognized as the most optimal primitive. If your geometry consists of several separate triangles that share vertices and edges, you might want to convert your data to triangle strips to improve performance.

To create triangle strips from separate triangles, you need to implement an algorithm to find and join adjacent triangles.

Code for doing this is available free on the Web. The Stripe package is one solution.

23 Extensions and Versions

23.010 Where can I find information on different OpenGL extensions?

The <u>OpenGL extension registry</u> is the central resource for OpenGL extensions. Also, the <u>OpenGL org Web page</u> maintains a lot of information on OpenGL extensions.

A list of extensions available on common consumer OpenGL devices is available.

Here's a similar list of extensions.

23.020 How will I know which OpenGL version my program is using?

It's commonplace for the OpenGL version to be named as a C preprocessor definition in gl.h. This enables your application to know the OpenGL version at compile time. To use this definition, your code might look like:

```
#ifdef GL_VERSION_1_2
    // Use OpenGL 1.2 functionality
#endif
```

OpenGL also provides a mechanism for detecting the OpenGL version at run time. An app may call glGetString(GL_VERSION), and parse the return string. The first part of the return string must be of the form [major-number].[minor-number], optionally followed by a release number or other vendor-specific information.

As with any OpenGL call, you need a current context to use glGetString().

23.030 What is the difference between OpenGL 1.0, 1.1, and 1.2?

In OpenGL 1.1, the following features are available:

- Vertex Arrays, which are intended to decrease the number of subroutine calls required to transfer vertex data to OpenGL that is not in a display list
- Polygon Offset, which allows depth values of fragments resulting from the filled primitives' rasterization to be shifted forward or backwards prior to depth testing
- ♦ Logical Operations can be performed in RGBA mode
- Internal Texture Formats, which let an application suggest to OpenGL a preferred storage precision for texture images
- Texture Proxies, which allow an application to tailor its usage of texture resources at runtime
- Copy Texture and Subtexture, which allow an application to copy textures or subregions of a texture from the framebuffer or client memory
- Texture Objects, which let texture arrays and their associated texture parameter state be treated as a single texture object

In OpenGL 1.2, the following features are available:

- Three–dimensional texturing, which supports hardware accelerated volume rendering
- BGRA pixel formats and packed pixel formats to directly support more external file and hardware framebuffer types

- Automatically rescaling vertex normals changed by the ModelView matrix. In some cases, rescaling can replace a more expensive renormalization operation.
- Application of specular highlights after texturing for more realistic lighting effects
- Texture coordinate edge clamping to avoid blending border and image texels during texturing
- Level of detail control for mipmap textures to allow loading only a subset of levels. This can save texture memory when high-resolution texture images aren't required due to textured objects being far from the viewer.
- Vertex array enhancements to specify a subrange of the array and draw geometry from that subrange in one operation. This allows a variety of optimizations such as pretransforming, caching transformed geometry, etc.
- The concept of ARB-approved extensions. The first such extension is GL_ARB_imaging, a set of features collectively known as the Imaging Subset, intended for 2D image processing. <u>Check for the extension string</u> to see if this feature is available.

OpenGL 1.2.1 adds a second ARB–approved extension, GL_ARB_multitexture, which allows multiple texture maps to be applied to a single primitive. Again, <u>check for the extension string</u> to use this extension.

23.040 How can I code for different versions of OpenGL?

Because a feature or extension is available on the OpenGL development environment you use for building your app, it doesn't mean it will be available for use on your end user's system. Your code must avoid making feature or extension calls when those features and extensions aren't available.

When your program initializes, it must query the OpenGL library for information on the OpenGL version and available extensions, and surround version– and extension–specific code with the appropriate conditionals based on the results of that query. For example:

```
#include <stdlib.h>
    ...
int gl12Supported;
gl12Supported = atof(glGetString(GL_VERSION)) >= 1.2;
...
if (gl12Supported) {
    // Use OpenGL 1.2 functionality
}
```

23.050 How can I find which extensions are supported?

A call to glGetString(GL_EXTENSIONS) will return a space–separated string of extension names, which your application can parse at runtime.

23.060 How can I code for extensions that may not exist on a target platform?

At runtime, your application can inquire for the existence of a specific extension using glGetString(GL_EXTENSIONS). Search the list of supported extensions for the specific extension you're interested in. For example, to see if the polygon offset extension interface is available, an application might say:

Your application can use the extension if it's available, but it needs a fallback plan if it's unavailable (i.e., some other way to obtain the same functionality).

If your application code needs to compile on multiple platforms, it must handle a development environment in which some extensions aren't defined. In C and C++, the preprocessor can protect extension–specific code from compiling when an extension isn't defined in the local development environment. For example:

```
#ifdef GL_EXT_polygon_offset
   glEnable (GL_POLYGON_OFFSET_EXT);
   glPolygonOffsetEXT (1., 1./(float)0x10000);
#endif /* GL_EXT_polygon_offset */
```

23.070 How can I call extension routines on Microsoft Windows?

Your application may find some extensions already available through Microsoft's opengl32.lib. However, depending on your OpenGL device and device driver, a particular vendor–specific extension may or may not be present at link time. If it's not present in opengl32.lib, you'll need to obtain the address of the extension's entry points at run time from the device's ICD.

Here's an example code segment that demonstrates obtaining function pointers for the ARB_multitexture extension:

```
/* Include the header that defines the extension. This may be a vendor-spe
    .h file, or GL/glExt.h as shown here, which contains definitions for al
    extensions. */
#include "GL/glExt.h"
/* Declare function pointers */
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB;
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB;
...
    /* Obtain the address of the extension entry points. */
    glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
    wglGetProcAddress("glActiveTextureARB");
    glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
    wglGetProcAddress("glMultiTexCoord2fARB");
```

After you obtain the entry point addresses of the extension functions you wish to use, simply call through them as normal function pointers:

```
/* Set texture unit 0 min and mag filters */
  (*glActiveTextureARB) (GL_TEXTURE0_ARB);
  glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameterf (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
...
```

```
/* Draw multi textured quad */
glBegin (GL_QUADS);
   (*glMultiTexCoord2fARB) (GL_TEXTURE0_ARB, 0.f, 0.f);
   (*glMultiTexCoord2fARB) (GL_TEXTURE1_ARB, 0.f, 0.f);
   glVertex3f (32.f,32.f, 0.f);
...
glEnd();
```

More information on wglGetProcAddress() is available through the MSDN documentation.

You might find it annoying to explicitly call through a function pointer. <u>A modified version of glext.h is</u> <u>available</u> that doesn't eliminate the function pointer, but hides it with the C preprocessor, allowing for more aesthetically pleasing code.

23.080 How can I call extension routines on Linux?

Like Microsoft Windows (and unlike proprietary UNIX implementations), an extension entry point may or may not be defined in the static link library. At run time, a Linux application must load the function's address, and call through this function pointer.

Linix uses the **OpenGL ABI**.

23.090 Where can I find extension enumerants and function prototypes?

See the **OpenGL** extension registry.

For specific files:

<u>glext.h</u> wglext.h glxext.h

glext.h is not a replacement for gl.h, it's a supplement. It provides interfaces for all extensions not already defined by the platform–specific gl.h. This is necessary for platforms that support multiple graphics drivers where the gl.h from a central source (e.g. Microsoft or XFree86) can't track functionality provided by more frequently updated vendor drivers.

24 Miscellaneous

24.010 How can I render a wireframe scene with hidden lines removed?

The preferred method is to render your geometry in two passes: first render it in fill mode with color set to the background color, then render it again in line mode. Use polygon offset so the lines over the polygons render correctly. <u>The polygon offset section</u> might be helpful to you.

Often you need to preserve a nonuniform background, such as a gradient fill or an image. In this case, execute the fill pass with glColorMask() set to all GL_FALSE, then perform the line pass as usual. Again, use polygon offset to minimize Z fighting.

24.020 How can I render rubber-band lines?

See this question in the Rasterization section.

24.030 My init code calls glGetString() to find information about the OpenGL implementation, but why doesn't it return a string?

The most likely cause of this problem is that a context hasn't been made current. An OpenGL rendering context must exist and be made current to a window for any OpenGL calls to function and return meaningful values.

24.039 Where can I find 3D model files?

As this has little to do with OpenGL, what follows is by no means an exhaustive list:

http://www.3dfiles.com/ http://www.3dcafe.org/ http://www.saturn-online.de/~cosmo/ http://www.swma.net/

You can make your own 3D models using any package you desire, and then loading the geometry file. <u>ModelMagic3D</u> is shareware and comes with source code. <u>GLScene</u> is also available.

24.040 How can I load geometry files, such as 3DS, OBJ, DEM, etc. and render them with OpenGL?

OpenGL, being a 3D graphics API, has no built–in support for reading application–specific file formats. If you're writing an application that needs to read a specific file type, you'll need to add code to support a particular file type.

Many OpenGL users already have written code to do this, and in some cases, the code is available on the Web. A few are listed here. If you can't find what you are looking for, you might try doing a Web search.

This file format information covers a variety of different file formats.

Okino's <u>PolyTrans</u> can convert most major <u>3D file formats</u> into OpenGL C code. Demos are available on their Web site.

<u>Crossroads</u> can import many file formats and output the data as C/C++ compilable data that is suitable for use with vertex arrays.

<u>3DWinOGL</u> is shareware that reads in any file format and returns OpenGL primitive data.

If you're using 3D Studio MAX, you should see an export format called ASE, which is ASCII (i.e., large file sizes), but is very easy to parse.

<u>The XGL file format</u> is intended to be capable of storing all OpenGL 3D information. An open source parser and a 3DS file converter are available.

Download the <u>GLUT</u> source distribution and look in progs/demos/smooth. The file glm.c contains routines for reading in Wavefront OBJ files.

glElite reads DXF, ASCII, and LightWave files. Information on glElite can be found at the following addresses: <u>http://www.helsinki.fi/~tksuoran/lw.html</u> and <u>http://www.cs.helsinki.fi/~tksuoran/glelite/</u>.

<u>3D Exploration</u> imports and exports several different file formats, including exporting to C/C++ source.

A 3DS import library in Delphi designed for use with OpenGL can be found here.

24.050 How can I save my OpenGL rendering as an image file, such as GIF, TIF, JPG, BMP, etc.? How can I read these image files and use them as texture maps?

To save a rendering, the easiest method is to use any of a number of image utilities that let you capture the screen or window, and save it is a file.

To accomplish this programmatically, you read your image with glReadPixels(), and use the image data as input to a routine that creates image files.

Similarly, to read an image file and use it as a texture map, you need a routine that will read the image file. Then send the texture data to OpenGL with glTexImage2D().

OpenGL will not read or write image files for you. To read or write image files, you can either write your own code, include code that someone else has written, or call into an image file library. The following links contain information on all three strategies.

This file format information covers a variety of different file formats.

The Independent JPEG Group has a free library for reading and writing JPEG files.

You can save your rendering as a JPEG image file, plus load JPEG and BMP files directly into OpenGL texture objects, using <u>the C++ mkOpenGLJPEGImage class</u>.

Source code for reading TGA files can be found here.

The gd library lets you create JPG and PNG files from within your program.

Imlib (search the "Download" section) is a wrapper library that allows a program to write out

JPEG, GIF, PNG, and TIFF files.

An image loader library in Delphi can be found here.

24.060 Can I use a BSP tree with OpenGL?

BSP trees can be useful in OpenGL applications.

OpenGL applications typically use the depth test to perform hidden surface removal. However, depending on your application and the nature of your geometry database, a BSP tree can enhance performance when used in conjunction with the depth test or when used in place of the depth test.

BSP trees also may be used to cull non-visible geometry from the database.

When rendering translucent primitives with blending enabled, BSP trees provide an excellent sorting method to ensure back-to-front rendering.

More information on BSP trees can be found at the BSP FAQ.

24.070 Can I use an octree with OpenGL?

Yes. Nothing in OpenGL prevents you from using an octree. An octree is especially helpful when used in conjunction with occlusion culling extensions (such as HP's GL_HP_occlusion_test).

24.080 Can I do radiosity with OpenGL?

OpenGL doesn't contain any direct support for radiosity, it doesn't prevent you from displaying a database containing precomputed radiosity values.

An application needs to perform its own radiosity iterations over the database to be displayed. After sufficient color values are computed at each vertex, the application renders the database as normal OpenGL primitives, specifying the computed color at each vertex. glShadeModel() should be set to GL_SMOOTH and lighting should be disabled.

24.090 Can I raytrace with OpenGL?

OpenGL contains no direct support for raytracing.

You might want to use raytracing to produce realistic shadows and reflections. However, you can simulate in many ways these effects in OpenGL without raytracing. See the <u>section on</u> <u>shadows</u> or the <u>section on texture mapping</u> for some algorithms.

You can use OpenGL as part of the ray intersection test. For example, a scene can be rendered with a unique color assigned to each primitive in the scene. This color can be read back to determine the primitive intersected by a ray at a given pixel. If the exact geometry is used in this algorithm, some aliasing may result. To reduce these aliasing artifacts, you can render bounding volumes instead.

Also, by changing the viewpoint and view direction, you can use this algorithm for

intersection testing of secondary rays.

A ray tracing application might also use OpenGL for displaying the final image. In this case, the application is responsible for computing the color value of each pixel. The pixels then can be rendered as individual GL_POINTS primitives or stored in an array and displayed via a call to glDrawPixels().

24.100 How can I perform CSG with OpenGL?

The <u>Opengl Programming Guide, Third Edition</u>, describes some techniques for displaying the results of CSG operations on geometric data.

The GLUT 3.7 distribution contains an example program called csg.c that may be informative.

24.110 How can I perform collision detection with OpenGL?

OpenGL contains no direct support for collision detection. Your application needs to perform this operation itself.

OpenGL can be used to evaluate potential collisions the same way it can <u>evaluate ray</u> <u>intersections</u> (i.e., the scene is rendered from the object's point of view, looking in the direction of motion, with an orthographic projection and a field–of–view restricted to the object's bounding rectangle.) Visible primitives are potential collision candidates. You can examine their Z values to determine range.

There's a free <u>library for collision detection called I COLLIDE</u> available that you might find useful.

24.120 I understand OpenGL might cache commands in an internal buffer. Can I perform an abort operation, so these buffers are simply emptied instead of executed?

No. After you issue OpenGL commands, inevitably they'll be executed.

24.130 What's the difference between glFlush() and glFinish() and why would I want to use these routines?

The OpenGL spec allows an implementation to store commands and data in buffers, which are awaiting execution. glFlush() causes these buffers to be emptied and executed. Thus, any pending rendering commands will be executed, but glFlush() may return before their execution is complete. glFinish() instructs an implementation to not return until the effects of all commands are executed and updated.

A typical use of glFlush() might be to ensure rendering commands are exected when rendering to the front buffer.

glFinish() might be particularly useful if an app draws using both OpenGL and the window system's drawing commands. Such an application would first draw OpenGL, then call glFinish() before proceeding to issue the window system's drawing commands.

24.140 How can I print with OpenGL?

OpenGL currently provides no services for printing. The OpenGL ARB has discussed a GLS stream protocol, which would enable a more common interface for printing, but for now, printing is only accomplished by system–specific means.

On a Microsoft Windows platform, ALT–PrintScreen copies the active window to the clipboard. (To copy the entire screen, make the desktop active by clicking on it, then use ALT–PrintScreen.) Then you can paste the contents of the clipboard to any 2D image processing software, such as Microsoft Paint, and print from there.

You can capture an OpenGL rendering with any common 2D image processing packages that provide a screen or window capture utility, and print from there.

Also, can print programatically using any method available on your platform. For example in Microsoft Windows, you might use glReadPixels() to read your window, write the pixel data to a DIB, and submit the DIB for printing.

24.150 Can I capture or log the OpenGL calls an application makes?

IBM has a product called ZAPdb that does this. It ships with many UNIX implementations, including IBM and HP. It was available on Windows NT in the past, but its current status is unknown. A non–IBM web page appears to have ZAPdb available for download.

Intel's GPT also supports this functionality.

There's a free utility called GLTrace2, which contains capture functionality similar to ZAPdb and GPT. <u>More info on GLTrace2 can be found here</u>.

In theory, you could code a simple library that contains OpenGL function entry points, and logs function calls and parameters passed. Name this library opengl32.dll and store it in your Windows system folder (first, be careful to save the existing opengl32.dll). This shouldn't be a difficult programming task, but it might be tedious and time consuming. This solution is not limited to Microsoft Windows; using the appropriate library name, you can code this capture utility on any platform, provided your application is linked with a dynamically loadable library.

24.160 How can I render red-blue stereo pairs?

The Viewing section contains a question on <u>creating a stereo view</u>, and has a link to information on creating anaglyphs. The basic idea, In OpenGL, is as follows:

- 1. glColorMask (GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE)
- 2. Assuming the red image is the left image, set the projection and model-view matrices for the left image.
- 3. Clear color and depth buffers, and render the left image.
- 4. glColorMask (GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE)
- 5. Set the projection and model–view matrices for the right image.
- 6. Clear color and depth buffers and render the right image.
- 7. Swap buffers.

There is a GLUT 3.7 demo that shows how to do this.

Appendix A Microsoft OpenGL Information

Submitted by Samuel Paik.

Windows Driver Development Kits

Preliminary Windows 2000 DDK

<u>Mini Client Driver</u>

<u>S3Virge</u>

[Sample Windows 2000 display driver supporting DirectDraw, Direct3D, OpenGL MCD, Video Port Extensions]

Windows Driver and Hardware Development

 OpenGL for 3D Color Graphics Programming [Summary of OpenGL support in Windows]

 Driver Licensing Program for OpenGL and Direct3D

 WHOL – Test Kits and Procedures [OpenGL Conformance tests are included in the display driver tests]

 GDI Display Drivers in Windows 2000

 GDI Display Drivers in Windows 2000

 Multimedia Components in Windows 95 and Windows 2000

 Implementing Display Control Panel Extensions in Windows 95 and Windows 98 [Notes on acceptible "Wait for Vblank" usage]

 Microsoft Releases New 3–D DDK [New ICD kit announcement including SGI OpenGL improvements—result of OpenGL truce with

SGI]

Fluff articles

Industry Solutions: OpenGL Update

[Says OpenGL is important to Microsoft and that OpenGL 1.2 support will likely be available in a future Windows 2000 Service Pack]

Insider: Fixing Color Distortions in Windows 98 3D Screen Savers

<u>Windows NT Workstation: Benchmark Results: Windows NT Workstation 4.0 Bests Unix Workstations in Two</u> <u>Industry–Standard Engineering Application Benchmarks</u>

Windows NT Workstation: Windows NT Workstation and Windows 95: Technical Differences

[Windows 95 acquired OpenGL with Service Pack 1]

POCKETPC: Here Comes GAPI!

[OpenGL and DirectX are too heavyweight for CE, so yet another "Game API"] <u>PressPass: Microsoft Delivers Performance–Leading Version of OpenGL</u>

[OpenGL 1.1 introduced for Windows 95 and Windows NT, 1.1 bundled with NT 4.0] <u>PressPass: Silicon Graphics and Microsoft Form Strategic Alliance To Define the Future of Graphics</u>

[Fahrenheit project announcement-goes with OpenGL truce]

<u>PressPass: Microsoft and Silicon Graphics Define Distribution And Support of OpenGL on the Windows</u> <u>Platform</u>

[Truce over OpenGL—goes with Fahrenheit announcement. New DDK to incorporate old ICD DDK with code from SGI OpenGL]

OpenGL 3–D Graphics

[OpenGL technology brief]

MSDN Library

Platform SDK

- <u>EMRGLSBOUNDEDRECORD</u> The EMRGLSBOUNDEDRECORD structure contains members for an enhanced metafile record generated by OpenGL functions. It contains data for OpenGL functions with information in pixel units that must be scaled when playing the metafile.
- <u>EMRGLSRECORD</u> The EMRGLSRECORD structure contains members for an enhanced metafile record generated by OpenGL functions, It contains data for OpenGL functions that scale automatically to the OpenGL viewport.
- OpenGL
 - ◆ <u>Legal Information</u>
 - ♦ Overview
 - ♦ Introduction to OpenGL
 - Primitives and Commands
 - · OpenGL Graphic Control
 - Execution Model
 - · Basic OpenGL Operation
 - · OpenGL Processing Pipeline
 - OpenGL Function Names
 - <u>Vertices</u>
 - Primitives
 - Fragments
 - <u>Pixels</u>
 - · Using Evaluators
 - <u>Performing Selection and Feedback</u>
 - Using Display Lists
 - · Managing Modes and Execution
 - · Obtaining State Information
 - · OpenGL Utility Library
 - ♦ Win32 Extensions to OpenGL
 - OpenGL on Windows NT, Windows 2000, and Windows 95/98
 - · Components
 - <u>Generic Implementation and Hardware Implementation</u>
 - · Limitations
 - · Guide To Documentation
 - · <u>Rendering Contexts</u>
 - <u>Rendering Context Functions</u>
 - Pixel Formats
 - Pixel Format Functions
 - · Front, Back, and Other Buffers
 - <u>Buffer Functions</u>
 - · Fonts and Text
 - Font and Text Functions
 - · <u>OpenGL Color Modes and Windows Palette Management</u>
 - Palettes and the Palette Manager
 - Palette Awareness

OpenGL FAQ and Troubleshooting Guide

- Reading Color Values from the Frame Buffer
- <u>Choosing Between RGBA and Color–Index Mode</u>
- <u>RGBA Mode and Windows Palette Management</u>
- <u>Color–Index Mode and Windows Palette Management</u>
- · Overlay, Underlay, and Main Planes
- · Sharing Display Lists
- · Extending OpenGL Functions
- · GLX and WGL/Win32
- <u>Using OpenGL on Windows NT/2000 and Windows 95/98</u>
 - Header Files
 - Pixel Format Tasks
 - ◆ <u>Choosing and Setting a Best–Match Pixel Format</u>
 - ◆ Examining a Device Context's Current Pixel Format
 - Examining a Device's Supported Pixel Formats
 - Rendering Context Tasks
 - ◆ Creating a Rendering Context and Making It Current
 - <u>Making a Rendering Context Not Current</u>
 - Deleting a Rendering Context
 - Drawing with Double Buffers
 - Drawing Text in a Double-Buffered OpenGL Window
 - Printing an OpenGL Image
 - Copying an OpenGL Image to the Clipboard
 - <u>Multithread OpenGL Drawing Strategies</u>
 - Using the Auxiliary Library
- Reference for Win 32 Extensions to OpenGL
- ◊ WGL and Win32 Functions and Structures
- ◊ Programming Tips
 - · OpenGL Correctness Tips
 - · OpenGL Performance Tips
- ♦ Reference
- Porting to OpenGL
 - ◊ Introduction to Porting to OpenGL for Windows NT, Windows 2000, and Windows 95/98
 - · Porting X Window System Applications
 - · Translating the GLX library
 - <u>Porting Device Contexts and Pixel Formats</u>
 - <u>GLX Pixel Format Code Sample</u>
 - <u>Win32 Pixel Format Code Sample</u>
 - · Porting Rendering Contexts
 - <u>GLX Rendering Context Code Sample</u>
 - <u>Win32 Rendering Context Code Sample</u>
 - · Porting GLX Pixmap Code
 - · Porting Other GLX Code
 - · <u>A Porting Sample</u>
 - <u>An X Window System OpenGL Program</u>
 - The Program Ported to Win32
 - · Porting Applications from IRIS GL
 - · Special IRIS GL Porting Issues
 - OpenGL Functions and Their IRIS GL Equivalents
 - ◊ IRIS GL and OpenGL Differences
- ♦ Glossary

♦ Appendix

♦ <u>About OpenGL</u>

OpenGL technical articles

<u> OpenGL 1.1</u>

[OpenGL 1.1 was first introduced into the Windows 9X line with Windows 95, OEM Service Release 2]

OpenGL I: Quick Start

This article describes GLEasy, a simple OpenGL program. OpenGL is a three–dimensional (3–D) graphics library included with the Microsoft® Windows NT® version 3.5 operating system. GLEasy is a Microsoft Foundation Class Library (MFC) application that provides a good starting point for investigations into the Windows NT implementation of OpenGL.

OpenGL II: Windows Palettes in RGBA Mode

If a program written for the Microsoft® Windows® operating system needs more than 16 colors and is running on an 8-bits-per-pixel (bpp) display adapter, the program must create and use a palette. OpenGL programs running on Windows NT® or (eventually) Windows 95 are no exception. OpenGL imposes additional requirements on the colors and their locations on the palette in RGBA mode. The articles "OpenGL I: Quick Start" and "Windows NT OpenGL: Getting Started" in the MSDN Library cover the basics of using OpenGL in a Windows-based program and are required reading for this article. Two sample applications, GLEasy and GLpal, accompany this article.

OpenGL III: Building an OpenGL C++ Class

This article discusses the development of a C++ class library for encapsulating OpenGLT code. The C++ class presented is for demonstration and educational purposes only. I will expand the class library for future OpenGL articles. The class library is not currently part of the Microsoft® Foundation Class Library (MFC), and there are no plans to add this class to MFC in the future. I assume that the reader has already read the first article in this series, "OpenGL I: Quick Start," in the MSDN Library. The class library is in the GLlib.DLL file included with this article. The EasyGL sample application, also included with this article, uses the classes in GLlib.DLL.

Color Index Mode

This article explores the Windows NTT implementation of OpenGLT color index mode. In color index mode, colors are specified as indexes into a palette instead of as levels of red, green, and blue. The EasyCI sample application (provided with this article) is a conversion of EasyGL that uses color index mode. EasyCI uses the GLlib.DLL, also included with this article.

OpenGL IV: Translating Windows DIBs

OpenGLT is a portable language for rendering three–dimensional (3–D) graphics. OpenGL does not understand Microsoft® Windows® device–independent bitmaps (DIBs); instead, it has its own format for representing images. This article explains how to translate a Windows DIB into a format usable with OpenGL. Some knowledge of the Windows DIB format and the Microsoft Foundation Class Library (MFC) is expected. The EasyDIB sample application and GLlib dynamic–link library (DLL) demonstrate the ideas presented in this article.

OpenGL VI: Rendering on DIBs with PFD DRAW TO BITMAP

The PFD_DRAW_TO_BITMAP pixel format descriptor flag allows OpenGLT applications to render on a Microsoft® Windows® device-independent bitmap (DIB). The resulting DIB can be manipulated to the full extent using the commands in the Windows graphics device interface (GDI). This article explains how you can render OpenGL scenes on DIBs with PFD_DRAW_TO_BITMAP. The EasyBit sample application demonstrates the techniques presented in the article.

OpenGL VII: Scratching the Surface of Texture Mapping

This article explains how to apply bitmaps to OpenGLT surfaces to give them a realistic appearance. The bitmaps are known as *textures* and can resemble wood, marble, or any other interesting material or pattern. The process of applying or mapping a texture to a surface is known as *texture mapping*.

The EasyTex and PicCube sample applications demonstrate the concepts discussed in this article. <u>OpenGL VIII: wglUseFontOutlines</u>

This article explains how to use the Win32® **wglUseFontOutlines** function. This function creates three–dimensional (3–D) characters based on a TrueType® font for use in OpenGLT–rendered scenes. The EasyFont sample application demonstrates using **wglUseFontOutlines**.

Windows NT OpenGL: Getting Started

OpenGL, an industry-standard three-dimensional software interface, is now a part of Microsoft® Windows NTT version 3.5. As a hardware-independent interface, the operating system needs to provide pixel format and rendering context management functions. Windows NT provides a generic graphics device interface (GDI) implementation for this as well as a device implementation. This article details these implementations, OpenGL/NT functions, and tasks that applications need to accomplish before OpenGL commands can be used to render images on the device surface.

CUBE: Demonstrates an OpenGL Application

CUBE is a simple OpenGLT application. It demonstrates how to integrate OpenGL with the MFC single document interface (SDI), and how OpenGL's resource contexts are used in conjunction with device contexts.

OPENGL: Demonstrates Using OpenGL

This sample creates a control that draws a spinning cube using the OpenGL graphics library. [Uses ATL: Active Template Library]

<u>OpenGL Without the Pain: Creating a Reusable 3D View Class for MFC</u> <u>DirectX 6.0 Goes Ballistic With Multiple New Features And Much Faster Code</u>

Get Fast and Simple 3D Rendering with DrawPrimitive and DirectX 5.0

February 97 Microsoft Interactive Developer Column: Fun and Games

[claims OpenGL will be based on Direct3D Immediate Mode in the future——I believe this work on this ended some time ago, may eventually be revived]

Poking Around Under the Hood: A Programmer's View of Windows NT 4.0

[What's new with Windows NT 4.0, including WGL (very misleading information)]

Windows NT Resource Kit: Registry Value Entries: Video Device Driver Entries

[OpenGL registry keys, among others]

Windows NT Resource Kit: Dynamic Link Library Files

[Annotated list of system DLLs]

DirectX Developer FAQ

[Notes that the DX7 Direct3D lighting model was changed to match OpenGL lighting]

Useful other articles

DIBs and Their Use

This article discusses the DIB (device–independent bitmap) concept from definition and structure to the API that uses it. Included is a small sample application that illustrates some of the most common methods of using DIBs to display and manipulate digital images. Functions discussed are **GetDIBits**, **SetDIBits**, **CreateDIBitmap**, **SetDIBitsToDevice**, **StretchDIBits**, and **CreateDIBPatternBrush**. This article does not discuss using palettes with DIBs.

Using DIBs with Palettes

This article discusses using palettes in conjunction with DIBs (device–independent bitmaps). It does not delve into involved uses of the Microsoft® WindowsT Palette Manager.

Creating Programs Without a Standard Windows User Interface Using Visual C++ and MFC

Microsoft® Visual C++T and the Microsoft Foundation Class Libraries (MFC) provided a very fast way to get a standard WindowsT-based application up and running. But what if you don't want the normal look and feel? Many games and educational applications have special user interface needs that can't be met with the standard Windows user interface. This article takes a look at creating a simple child's coloring game that uses only a single window and has no window border, caption,

buttons, cursor, or any other recognizable elements of a Windows user interface.

Knowledge Base

Current

<u>Q254265 – 'Advanced' Button Under 'Display' Does Not Work After Installation of Windows NT 4.0 Drivers</u> <u>in Windows 2000</u>

[Windows 2000] After you upgrade from Microsoft Windows NT 4.0 to Microsoft Windows 2000, or after you install Windows NT 4.0 drivers in Windows 2000, and you click the Advanced button on the Settings tab under Display in Control Panel, you may receive an error message.

<u>Q253521 – INFO: OpenGL Drivers</u>

OpenGL drivers have traditionally been provided by the hardware vendors who provide the 3D adapter in your computer.

<u> 0247438 – OpenGL Support Not Available on nVidia TNT2 Card in Microsoft Windows 2000</u>

[Windows 2000] When you attempt to play a game that requires support for the OpenGL standard (for three–dimensional graphics display) on a Microsoft Windows 2000–based computer, the game does not run. [ed note: Microsoft does not ship display drivers with OpenGL support with Windows 2000]

<u>O240896 – OpenGL Program May Cause an Invalid Page Fault Error Message if the Window is Moved or</u> <u>Resized</u>

[Windows 95, 98, 98SE] When you move or resize a window, a program that uses OpenGL may perform an illegal operation, and then shutdown. For example, Microsoft Internet Explorer may generate an invalid page fault if a Java tool using OpenGL is running, and the window displaying the OpenGL graphic content is moved. Also, the following message may be generated in the Details section of the Application error dialog box:

<u>Q233390 – BUG: First Chance Exceptions When Calling ChoosePixelFormat</u>

[Windows 95, 98] The following error is displayed in the debug window of Visual C++: First–chance exception in myapp.exe (GDI32.DLL): 0xC0000005: Access Violation.

<u>Q228099 – PRB: wglUseFontOutlines Does Not Handle DBCS</u>

[Windows 98, NT 4.0] On Windows 98, the OpenGL function wglUseFontOutlines does not work with DBCS or UNICODE strings. On Windows NT, UNICODE strings work; however, DBCS strings do not.

<u> 0227279 – OpenGL Screen Saver Prevents Power Management Standby Mode</u>

[Windows 2000] When you configure your computer to use an OpenGL screen saver and the System Standby feature in Advanced Power Management (APM), your computer may not start the Standby mode.

Glide API Features Disabled on Video Adapter

[Windows NT 4.0; I don't see why this doesn't affect Windows 9X or Windows 2000. The description is confused] After you install Windows NT 4.0 Service Pack 4 on a computer with a proprietary 3Dfx function library file (such as the 3dfxgl.dll file installed during the installation of id Software's Quake II), you may not be able to access your video adapter's support for 3Dfx graphics.

Windows 98 Components for Typical, Portable and Compact Setup

[Lists components installed, OpenGL is not installed in "Compact" installation]

<u>Q176752 – Glen.exe Shows How to Enumerate Pixel Formats in OpenGL</u>

The GLEnum sample provides a demonstration of how to enumerate pixel formats and method for checking the available pixel formats provided on your machine. The GLEnum sample is included in Glen.exe.

<u>GLEN.EXE: SAMPLE: Pixel Format Enumeration in OpenGL Demo</u> <u>Q169954 – INFO: Layer Planes in OpenGL</u> Layer Planes are a new feature in the Microsoft implementation of OpenGL 1.1. Before using OpenGL layer planes, there are several new functions and some driver dependency issues that you should be aware of.

<u>Q160817 – Demonstrates OpenGL Texture–Mapping Capabilities</u>

GLTEXTUR.EXE provides a demonstration of how to use a Device–independent Bitmap (DIB) as a texture–map for OpenGL by pasting a DIB (chosen by the user) onto three different OpenGL objects. <u>GLTEXTUR.EXE: SAMPLE: Demonstrates OpenGL Texture–Mapping Capabilities</u>

<u>Q154877 – OpenGL 1.1 Release Notes & Components</u>

Opengl95.exe contains the release notes for OpenGL version 1.1 for Windows 95 and all of the components associated with OpenGL such as the DLL, library, and include files.

Note that Windows 95 OSR2, Windows 98, and Windows NT already include OpenGL with the O.S., so this download is not necessary (or recommended) for those platforms

OPENGL95.EXE

<u> 0152001 – GLLT.EXE Demonstrates Simple Lighting in OpenGL</u>

The GLLight sample provides a demonstration of how the various light settings effect an OpenGL scene. The initial scene is simply a single white sphere with a single blue light (GL_LIGHT0) shining on it.

<u> 0151489 – INFO: When to Select and Realize OpenGL Palettes</u>

An OpenGL application must select and realize its palette before setting the current rendering context with wglMakeCurrent.

<u>Q148301 – GLTex Demos How to Use DIBs for Texture Mapping</u>

The GLTex sample provides a demonstration of how to use a DIB (device– independent bitmap) as a texture–map for OpenGL by pasting a DIB (chosen by the user) onto all sides of a three–dimensional cube. [Appears to have been superceded by Q160817, code no longer here.]

<u>Q139967 – GLEXT: Demo of GL WIN swap hint & GL EXT vertex array</u>

The GLEXT sample illustrates how to use the GL_WIN_swap_hint extension to speed up animation by reducing the amount of repainting between frames and how to use GL_EXT_vertex_array extension to provide fast rendering of multiple geometric primitives with one glDrawArraysEXT call. It also shows how to use glPixelZoom and glDrawPixels to display an OpenGL bitmap.

0139653 – PRB: Antialiased Polygons Not Drawn in OpenGL Antipoly Sample

The antipoly sample in OpenGL SDK BOOK directory is unable to draw antialised polygons with the generic implementation of Windows NT and Windows 95 OpenGL.

<u>Q136266 – Demonstration of OpenGL Material Property and Printing</u>

The GLBMP sample illustrates how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors; the shininess; and the color of any emitted lights. This sample also demonstrates how to print an OpenGL image by writing the OpenGL image into a DIB section and printing the DIB section. The current version of Microsoft's implementation of OpenGL in Windows NT does not provide support for printing. To work around this current limitation, draw the OpenGL image into a memory bitmap, and then print the bitmap.

GLBMP.EXE: Sample: OpenGL Material Property & Printing

<u>Q131130 – HOWTO: Set the Current Normal Vector in an OpenGL Application</u>

[Information on using the cross product to obtain a normal vector for a polygon]

<u>Q131024 – Drawing Three–Dimensional Text in OpenGL Appliations</u>

GDI operations, such as TextOut, can be performed on an OpenGL window only if the window is single–buffered. The Windows NT implementation of OpenGL does not support GDI graphics in a double–buffered window. Therefore, you cannot use GDI functions to draw text in a double–buffered window, for example. To draw text in a double–buffered window, an application can use the wglUseFontBitmaps and wglUseFontOutlines functions to create display lists for characters in a font, and then draw the characters in the font with the glCallLists function. The **wglUseFontOutlines** function is new to Windows NT 3.51 and can be used to draw 3–D characters of TrueType fonts. These characters can be rotated, scaled, transformed, and viewed like

any other OpenGL 3–D image. This function is designed to work with TrueType fonts. The GLFONT sample shows how to use the **wglUseFontOutlines** function to create display lists for characters in a TrueType font and how to draw, scale, and rotate the glyphs in the font by using glCallLists to draw the characters and other OpenGL functions to rotate and scale them. You need the Win32 SDK for Windows NT 3.51 to compile this sample, and you need to incorporate **wglUseFontOutlines** in your own application. You also need Windows NT 3.51 to execute the application.

GLFONT.EXE: Sample: Drawing 3-D Text in an OpenGL App

<u> 0127071 – MFCOGL a Generic MFC OpenGL Code Sample</u>

Microsoft Windows NT's OpenGL can be used with the Microsoft Foundation Class (MFC) library. This article gives you the steps to follow to enable MFC applications to use OpenGL.

MFCOGL.EXE: Code Sample Demonstrates Using OpenGL with MFC

<u>Q128122 – Implementing Multiple Threads in an OpenGL Application</u>

It is possible to create multiple threads in an OpenGL application and have each thread call OpenGL functions to draw an image. You might want to do this when multiple objects need to be drawn at the same time or when you want to have certain threads perform the rendering of specific types of objects.

GLTHREAD.EXE: SAMPLE: Using Multiple Threads in OpenGL App

<u>Q126019 – PRB: Most Common Cause of SetPixelFormat() Failure</u>

SetPixelFormat() fails with incorrect class or window styles. [I'm not convinced this is the **most** common cause today.]

Q124870 – XFONT.C from SAMPLES\OPENGL\BOOK Subdirectory

XFONT.C from the SAMPLES\OPENGL\BOOK subdirectory is not in the MAKEFILE, and subsequently is never built.

OPENGL3.EXE: MSJ Source: Feb '95: OPENGL3.EXE

[The associated KB article Q124/2/06 has disappeared. This code apparently went with the Microsoft Systems Journal "Understanding Modelview Transformations in OpenGL for Windows NT"] *O124034 – OpenGL Interface in Windows NT 3.5*

<u> <u>J124034 – OpenGL Interface in Windows IVI 5.5</u> This article defines and explains the OpenGL interface that is a</u>

This article defines and explains the OpenGL interface that is available and can be implemented in Windows NT version 3.5.

<u> 0121381 – Microsoft Systems Journal: November 1994</u>

This article lists the filenames and Snumbers for files available from online services that contain the source code described in articles published in the November 1994 issue of the "Microsoft Systems Journal."

CUBES.EXE: MSJ Source: Nov, 1994 cubes.exe

[This code apparently went with the Microsoft Systems Journal article introducing OpenGL with Windows NT 3.5: "3–D Graphics for Windows NT 3.5. Introducing the OpenGL Interface, Part II."] <u>0121282 – OPENGL Screen Savers May Degrade Server Performance</u>

If OPENGL screen savers are used on a Windows NT Server, network server performance (the Server's responsiveness to clients) may be degraded while the screen saver is running.

OPENGL.EXE: MSJ Source: Oct, 1994 opengl.exe

[Associated KB article Q119/8/62 appears to have disappeared. This code apparently went with the Microsoft Systems Journal article introducing OpenGL with Windows NT 3.5: "3–D Graphics for Windows NT 3.5. Introducing the OpenGL Interface, Part I."]

Archive

<u>O224792 – List of Bugs Fixed in Windows NT 4.0 Service Pack 1, 2, and 3</u> <u>Err Msg: STOP 0x00000050 PAGE FAULT IN NONPAGED AREA</u>

[Windows NT 4.0] When you run NetMeeting with sharing enabled, you may receive the following error message on a blue screen if you restart your computer and start NetMeeting again:

<u>Q191359 – SMS: Windows 95 OpenGL Screen Saver May Cause Computer to Stop</u>

[Windows 95 OSR2] Computers that are running Microsoft Windows 95 may lose their ability to safely shut down after the OpenGL or Mystify Your Mind screen saver is started and stopped several times. This may occur on computers that have the ATI 64 and ATI Rage Series video adapters installed.

<u> 0189979 – OpenGL–Based Programs Do Not Work After Upgrade to Windows 98</u>

[Windows 98] After you upgrade to Windows 98, your OpenGL–based programs may no longer work correctly, or may not work at all.

<u>Q166334 – OpenGL Access Violation on Windows NT Version 4.0</u>

[Windows NT 4.0] Under heavy stress, OpenGL applications may experience access violations. Also, OpenGl Line and Polygon texture clipping functions may fail when fogging is enabled.

<u>Q166257 – Applications Using OpenGL Cause Access Violation in OPENGL.DLL</u>

[Windows NT 4.0] A multi-threaded or multi-windowed application that uses OpenGL may cause an access violation in the Opengl.dll library.

<u>Q166198 – Display Color Problem with OpenGL Applications in Windows NT 4.0 Service Pack 2</u>

[Windows NT 4.0 SP2] After you apply Windows NT 4.0 Service Pack 2, coloring problems may occur with OpenGL applications where the wrong colors are drawn in a wide variety of situations. [See Q163677]

<u> 0164158 – OpenGL Diffuse Settings Revert to Default</u>

[Windows NT 4.0] When using OpenGL with Windows NT, the diffuse parameter changes back to the default when the color material changes from AMBIENT_AND_DIFFUSE to AMBIENT.

<u>0163677 – BUG: OpenGL Color Problems Using Service Pack 2 for Win NT 4.0</u>

[Windows NT 4.0 SP2] When you use Service Pack 2 for Windows NT 4.0, various coloring problems may arise that are not present in previous versions. The coloring problems involve drawing the wrong colors in a variety of situations.

GLSP2FIX.EXE: BUG: OpenGL Color Problems Using Service Pack 2 for Win NT 4.0 0160651

[pre–Windows NT 4.0 SP2] An application that uses OpenGL may crash with an exception 0xC0000090.

<u>Q159129 – OpenGL Access Violation with Invalid OpenGL Context</u>

[Pre–Windows NT 4.0 SP2] The API gluGetString causes an access violation and affects OpenGL operations.

<u>Q156473 – BUG: Windows NT Version 4.0 Bug List – GDI</u>

[Windows NT 4.0. Known bugs at time of release]

<u>Q152841 – Windows NT 4.0 Service Pack 3 Readme.txt File (40-bit)</u>

<u>Q147798 – Windows NT 4.0 Service Pack 3 Readme.txt File (128-bit)</u>

Access Violation in glsbCreateAndDuplicateSection API on PowerPC

[Windows NT 3.51 for PowerPC] When you install a OpenGL client video driver on your PowerPC computer running Windows NT and you run an OPENGL program, for example, the Windows NT Pipes screen saver, an access violation occurs in the glsbCreateAndDuplicateSection application programming interface (API).

<u>Q134893 – 3D OpenGL Screen Saver Restores Windows NT 3.51 Help</u>

[Windows NT 3.51] When you return to your desktop from any of Windows NT 3D OpenGL screen savers, any minimized Windows NT 3.51 Help files that use the Windows 95 Help engine are restored to full size.

<u>Q134765 – Unknown Software Exception When Application Calls OpenGL</u>

[Windows NT 3.51] An unknown software exception occurs when applications call OpenGL. When Windows NT attempts to shutdown the computer, a blue screen appears.

<u>Q133322 – List of Confirmed Bugs in Windows NT Version 3.51</u>

<u>Q133220 – List of Confirmed Bugs in Windows NT Version 3.5</u>

<u> 0132866 – DOCERR: Printing an OpenGL Image</u>

The documentation relating to printing an OpenGL image in the Win32 SDK versions 3.5, 3.51, and 4.0 is incorrect. The current version of Microsoft's implementation of OpenGL in Windows NT does not provide support for printing. More specifically, an application cannot call wglCreateContext or wglMakeCurrent on a printer device context.

<u>Q132748 – Choosing a Workstation OS: Windows 95/Windows NT Workstation</u> <u>Q128531 – README.TXT: Windows NT Version 3.51 U.S. Service Pack</u>

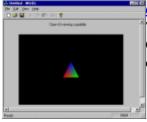
Snow/White Noise with Mach 32 at 1024x768 – 65536 colors

[Windows NT 3.5] When you use the ATI Mach 32 video adapter driver included with Windows NT version 3.5, white haze (also known as snow) may appear when you move windows on the desktop. This problem can also occur when you use the 3D Pipes (OpenGL) screen saver.

<u> 0126128 – Message Popup Changes Color When Using OpenGL Screen Saver</u>

[Windows NT 3.5] When you run Windows NT with a 800 x 600 (256 color) or 1024 x 768 (256 color) video driver and test an OpenGL screen–saver, the Title Bar and OK button in the Messenger Service dialog box are red.

Appendix B Source Code Index



<u>GlView.zip</u>

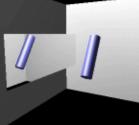
This code demonstrates use of OpenGL and MFC. OpenGL is rendered into a CStatic form control. For more information on using OpenGL with MFC, see questions <u>5.150</u>, <u>5.160</u>, <u>5.170</u>, and <u>5.180</u>.

<u>lookat.cpp</u>



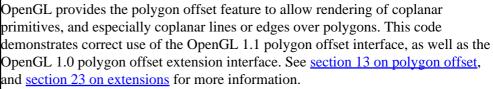
Many new OpenGL programmers are also new to linear algebra, and manipulating matrices can present a challenge. This code shows how to create a transformation matrix that will make an object point in a given direction. <u>Section 9 on</u> transformations may also be helpful.

<u>mirror.c</u>



Stencil planes can be used to render mirrors in OpenGL, but because many low–end graphics devices do not support them efficiently, using stencil planes is not practical. This code demonstrates how to use the depth buffer to render mirrors. An overview of the technique can be found in <u>question 9.170</u>.

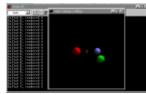
<u>pgonoff.c</u>





twopass.cpp

Since GL_MODULATE texture environment mode multiplies color values, obtaining white specular highlights on texture mapped objects requires special techniques. This code demonstrates a portable two–pass method, and also shows use of HP's pre–specular extension on platforms that support it. <u>Question</u> 21.040 discusses the issues involved in specular highlights on texture mapped objects.



<u>viewcull.c</u>

OpenGL clips geometry to the view volume a single vertex at a time. For optimum performance, an application must "bulk cull" large amounts of geometry. This code demonstrates how to obtain object space plane equations for the view volume, and how to clip test bounding boxes against them. <u>Section 10 on clipping</u> contains more information.

Here are few questions I expect to be frequently asked about GLUT 3.7. First, here are tag-line summaries of the question subject matter.

- 1. Problems building GLUT.
- 2. More GUI features.
- 3. New with GLUT 3.0.
- 4. GLUT for NT.
- 5. GLUT for OS/2.
- 6. <u>GLUT for Power Mcintosh.</u>
- 7. GLUT 3.0 incompatibilities.
- 8. <u>GLUT and Motif.</u>
- 9. <u>aux conversion to GLUT.</u>
- 10. SGI N32 and 64-bit support.
- 11. FORTRAN and GLUT.
- 12. Sophisticated input devices.
- 13. <u>GLUT and Open Inventor.</u>
- 14. GLUT, Sun, and Overlays.
- 15. The GLUT stroke font.
- 16. My book on GLUT.
- 17. GLUT and Microsoft portability.
- 18. GLUT and networking.
- 19. Asking GLUT questions.
- 20. Free OpenGL.
- 21. <u>GLUT overlay example code.</u>
- 22. BadMatch errors running GLUT programs.
- 23. <u>New with GLUT 3.1</u>.
- 24. <u>Shared libraries for Linux</u>25. <u>New in GLUT 3.2.</u>
- 26. GLUT API man pages.
- 27. Fast window repair for Mesa.
- 28. Advanced GLUT example .rgb image files.
- 29. IRIX 6.3 and 6.4 fast atoms support issues for older IRIX releases.
- 30. GLUT for the Power Macintosh.
- 31. New in GLUT 3.4
- 32. Cosmo3D beta and GLUT problem.
- <u>New in GLUT 3.5.</u>
 <u>Using the precompiled GLUT DLLs with Borland compilers.</u>
- 35. Using GLUT with C++.
- 36. How do you avoid the Console window appearing when you compiler a Win32 GLUT application with Microsoft compilers?
- 37. What is new in GLUT 3.6?
- 38. Why am I get build problems dealing with "glXChannelRectSyncSGIX" on an SGI O2 running IRIX 6.3?
- 39. Floating point exceptions using GLUT with Microsoft OpenGL 1.1 and compiling with Borland compilers.
- 40. Linking problems using GLUT with SGI OpenGL for Windows and compiling with Borland compilers.
- 41. What is GameGLUT?

Q1: I've tried to use the "mkmkfiles.imake" script to generate Makefiles so I can build GLUT, but it doesn't seem to work.

A1: While Imakefiles are supposted to be system independent (hence the "I"), the commands to translate Imakefiles into Makefiles varies from system to system. The X Consortium provides a command called "xmkmf", but vendors do not put this command in a consistent place. The "mkmkfiles.imake" script tries its best to generate Makefiles, but may get confused by different vendors configurations that I am not aware of.

It is also possible the imake configuration files (typically located at /usr/lib/X11/config) are buggy or from a very old version of X.

SGI users can benefit from using the "mkmkfile.sgi" script that uses SGI's parallel make, though "mkmkfiles.imake" should work too.

Q2: GLUT needs improved menus, dialog boxes, scrollbars, text entry fields, etc. to be useful to me?

A2: GLUT does not pretend to be a full-featured graphical user interface toolkit.

You _could_ write these sorts of GUI objects using GLUT and OpenGL if you needed to. The other alternative is to use Motif or whatever full featured toolkit you have.

Q3: What new things are in GLUT 3.0?

A3: See README.glut3 or read The OpenGL Utility (GLUT) Programming Interface document.

Q4: Is there a version of GLUT for Windows NT or Windows 95.

A4: Nate Robins and Layne Christensen at Evans & Sutherland has been working on a freely distributable version of <u>GLUT for Windows 95 and NT (European mirror</u>). His efforts are directed at porting GLUT 3.3.

Q5: Is there a version of GLUT for OS/2?

A5: Yes. I believe a version based on GLUT 2.x is distributed on an OS/2 OpenGL developer's CD-ROM.

Q6: Is there a version of GLUT for the Power Mcintosh?

A6: Was told by Template Graphics that an incomplete version of GLUT had been developed for their OpenGL product for the Power Mcintosh. I am not sure if it was ever completed or made available.

Q7: I'm hesitant about upgrading to GLUT 3.0 since I've got things working will with GLUT 2.3. Is the transition painful?

A7: I do not believe so. There are two changes worth noting that _may_ affect programs you have written.

First, you need a display callback registered before your display your windows on the screen. It did not make sense for this to not be true. In all likelihood, this should not affect your GLUT programs if they written well.

Second, you can no longer change, create, or destroy menus while pop-up menus are in use. Before, you could do this, but it meant a menu might be changed while in use. It was near impossible to describe what should happen in the case of menus being changed while in use that was likely to be portable to the way other window systems handled menus, so I made the practice illegal.

You can register a menu status callback to know when menus become used and unused to avoid changing menus while they are in use.

For more details about what has changed, see the CHANGES file.

Q8: So how do I use GLUT and Motif together?

A8: You don't. To make GLUT simple and easy-to-program, GLUT supplies its own event processing loop. This makes it nearly impossible to combine GLUT and Motif. If you want Motif, you probably want a full-featured toolkit, and you ship skip GLUT and implement your application directly in Motif.

Q9: I have a bunch of simpe OpenGL programs using the aux toolkit described in the OpenGL Programming Guide (the "red" book). Is there an easy way to convert them to GLUT?

A9: In the progs/redbook directory, there is a script named aux2glut.sed It will give you a good start at converting simple aux calls to their GLUT equivalents. It is a good start, but you'll still have to hand edit some things.

Here's a usage example:

sed -f aux2glut.sed < aux_prog. > glut_prog.c

Q10: I have IRIX 6.2 (or 6.1) and I'd like to write GLUT programs run in true 64-bit and/or benefit from the recent, faster MIPS processors. How do I build GLUT to support these newer application binary interfaces (ABIs)?

A10: See README.irix6

Q11: I'd like to write FORTRAN programs using GLUT and OpenGL. How do I use GLUT with FORTRAN?

A11: GLUT does have a FORTRAN language binding.

For instructions for building a binding library for Silicon Graphics workstations, see README.fortran

If you want to use GLUT and OpenGL or Mesa on with Fortran on non-SGI systems, I recommend that you check, William Mitchell's <u>f90gl home page</u>.

Q12: I'd like to use the sophisticated input devices that GLUT supports. What should I know about this?

A12: GLUT uses the X Input extension to talk to these devices. Because the X Input extension gives a framework for supporting input devices, but does not manadate how particular devices are supported, it is possible that each vendor supports the same input devices differently.

GLUT as implemented supports SGI's means of advertising the tablet, dial & button box, and Spaceball devices. I am not sure how other vendors support these devices. For the details of SGI's support for these devices, see README.xinput Since there is no benefit in each vendor supporting these same devices in a different an incompatible way, I encourage other vendors to implement their devices in this same manner.

Q13: Can I use GLUT and Open Inventor?

A13: Yes. See the README.inventor file. Also, some source code examples can be found at progs/inventor

Because the Open Inventor development environment is not supported on all systems, the Inventor example programs are not built by default, and the Makefile there only support SGI systems.

Q14: I have Sun workstation, and it is supposed to support overlays. So why does GLUT not use them?

A14: GLUT uses the SERVER_OVERLAY_VISUALS convention that advertises overlay visuals. Most major workstation vendors support this convention (DEC, HP, IBM, SGI), but Sun does not.

Q15: The stroke font used for GLUT looks familar. Where did it come from?

A15: The data for the "stroke roman" font is lifted from the X11R5 PEX sample implementation.

Q16: I read in the NOTICE file that you are writing a book on programming OpenGL for the X Window System. When will it be available?

A16: At SIGGRAPH '96 or possibly before that.

Q17: You mention an unnamed bu "very large window system software vendor" as the reason portable GLUT programs should not directly include <GL/gl.h> and <GL/glu.h> directly. What's the vendor and what are the details?

A17: Microsoft. It's version of $\langle GL/gl.h \rangle$ requires $\langle windows.h \rangle$ to be included before $\langle GL/gl.h \rangle$ can be included because of Microsoft function declaration conventions. Sigh.

Q18: I want my GLUT program to read and send information over a socket to some other program. How do I do this in in GLUT?

A18: You can not do it currently. I am considering such support for a possible GLUT 4.0. I'd like to have a portable solution.

What you'd like is a callback that would tell you when a socket is ready for reading and writing. I'm hoping to find a way to support this in an operating system independent manner. Does anyone know of a good portable interface for networked bytestream connections?

For now, you've got the source code to GLUT and you could hack it into GLUT for whatever particular interface your operating system provides.

Q19: Where's the best place to ask questions about GLUT or OpenGL? Can I just email them to you?

A19: While I may try to return email if I have time, the best place is the comp.graphics.api.opengl newsgroup. This gives a lot more people a chance to answer your question and you'll probably get an answer much faster than sending me email. Plus, I may not know the answer though someone on the "net" may know it.

Q20: My workstation doesn't have OpenGL. Where can I get a free copy to use with GLUT?

A20: OpenGL is licensed by Silicon Graphics and is not available as "free" or "public domain" software, but workstation vendors typically bundle OpenGL software with their workstation. However, there is a package called Mesa written by Brian Paul at the University of Wisconsin that implements the OpenGL API. (To be branded as "OpenGL", an implementation must be licensed *and* pass the Architectural Review Board's conformance suite, so Mesa is not an official "OpenGL" implementation.) Mesa does work with GLUT.

Q21: I hear GLUT 3.0 has overlay support. Where is an example?

A21: Look at progs/examples/zoomdino.c for an example of using overlays for rubber-banding and display of a help message, both in the overlays. Also, test/over test.c exercises all of the overlay routines.

Q22: I get BadMatch X protocol errors when I run GLUT programs. What gives?

A22: There is a bug in the Solaris 2.4 and 2.5 implementation of XmuLookupStandardColormap (fixed in Solaris 2.6). When you compile GLUT on Solaris 2.4 or 2.5, please apply the following patch and compile with _ DSOLARIS_2_4_BUG to workaround the problem. To do this, edit the glut/lib/glut/Makefile and add _ DSOLARIS_2_4_BUG to the CFLAGS macro. See the comment in the patch below. This code is already in GLUT 3.1 and later.

```
*** glut_win.c Wed Apr 24 14:06:08 1996
--- glut_win.c.bad
                      Wed Apr 24 14:03:58 1996
*** 398,414 ****
   case TrueColor:
   case DirectColor:
                         /* NULL if RGBA */
      *colormap = NULL;
 #ifndef SOLARIS_2_4_BUG
     /* Solaris 2.4 has a bug in its XmuLookupStandardColormap
          implementation. Please compile your Solaris 2.4 version
_
         of GLUT with -DSOLARIS_2_4_BUG to work around this bug.
_
         The symptom of the bug is that programs will get a
         BadMatch error from X_CreateWindow when creating a GLUT
         window because Solaris 2.4 creates a corrupted
         RGB_DEFAULT_MAP property. Note that this workaround
_
         prevents Colormap sharing between applications, perhaps
_
         leading unnecessary colormap installations or colormap
         flashing. */
      status = XmuLookupStandardColormap(___glutDisplay,
        vi->screen, vi->visualid, vi->depth, XA_RGB_DEFAULT_MAP,
         /* replace */ False, /* retain */ True);
--- 398,403 ----
*****
*** 423,429 ****
              return;
            }
- #endif
      /* If no standard colormap but TrueColor, just make a
         private one. */
      /* XXX Should do a better job of internal sharing for
--- 412,417 ----
```

Q23: What is new in GLUT 3.1?

A23: GLUT 3.1 is largely a maintence release. There are some new programs, a few minor GLUT library bug fixes, but mostly GLUT 3.1 is to make sure GLUT builds cleanly on various platforms like SunOS, HP/UX, Solaris, and Linux. See the CHANGES file included in the distribution for more details.

Q24: How do I make Linux shared libraries for GLUT?

A24: Peter F. Martone (<u>pmarton@mailbox.bgsu.edu</u>) has written some instructions for making a Linux shared library for GLUT. You can grab the instructions for doing so from <u>http://pizza.bgsu.edu/cgi-bin/cgiwrap/~pmarton/makeMainIndex</u>

Q25: New in GLUT 3.2.

A25: Like GLUT 3.1, GLUT 3.2 is a maintence release. Along with bug fixes to the core GLUT library, many new GLUT example programs have been added. The portability of the examples has been improved so that most should build using Windows 95 and NT. Also, GLUT API man pages are now included. See the CHANGES file included in the distribution for more details.

Q26: GLUT API man pages.

A26: Please see the README.man file for details. The easiest way for SGI users to get the man pages is to install the "glut_dev.man.glut" subsystem included with the pre-compiled SGI GLUT images.

Q27: Fast window repair for Mesa.

A27: The GLX specification states that the state of a window's back color buffer after a glXSwapBuffers is undefined. However, the freeware Mesa implementation of the OpenGL API always leaves the back buffer with its previous contents (ie, it simply "copies" the back buffer contents to the front buffer).

Because Mesa lacks hardware acceleration and is often slow to redraw a window, this presents the opportunity to speed redrawing a window damaged by window system interactions by simply calling glXSwapBuffers again.

If you set the MESA_SWAP_HACK environment variable, GLUT 3.2 will try to repair double buffered windows not otherwise needing a redisplay because of glutPostRedisplay by calling glXSwapBuffers when Mesa is the OpenGL implementation being used and the last display callback called glutSwapBuffers.

In general, this means if you see MESA_SWAP_HACK when using Mesa, double buffered GLUT programs will redraw very quickly after being damaged but still operate well if they've been correctly written to use glutPostRedisplay to trigger application required redraws.

I encourage all Mesa users to set the MESA_SWAP_HACK environment variable.

Q28: Advanced GLUT example .rgb image file.

A28: Yes, the image files these examples use are large and were seperated out from the main GLUT source code distribution. Get the glut_data.tar.gz file from where you got your GLUT distribution. Untar these data files over your glut distribution so the "data" directory is at the same level as "progs". Then do a "make links" in the progs/advanced directory to make symbolic links.

See the progs/advanced/README file for more details.

Q29: Why doesn't GLUT programs compiled on IRIX 6.4 or 6.3 work earlier releases?

A29: First, SGI never guarantees that an executable built on a later IRIX release will work on an earlier release. Sometimes it works; more often than not it does not. GLUT takes advantage of a new X optimization in IRIX 6.3 called "fast atoms". This optimization lets X clients determine common atom values without an X server round-trip. This helps X performance.

If you compile the GLUT library on an IRIX 6.3 or IRIX 6.4 machine, the library will support fast atoms. This will mean that if you run executables linked against the "fast atom enabled" version of the GLUT library, you'll get a run-time link error saying something like:

17062:qlut_example: rld: Fatal Error: attemped access to unresolvable symbol in projtex:

_XSGIFastInternAtom

Do not be alarmed. If you want, you can recompile the GLUT library with the -DNO_FAST_ATOMS and get a version of the library that doesn't have the support so that GLUT executables built with a library compiled without "fast atoms" can work on earlier IRIX releases. *Note that even if you do compile with -DNO_FAST_ATOMS, there is still no guarantee that an IRIX executable compiled on a newer release will actually work on an older release (but at least you'll have a chance!*).

Note that the precompiled images lack "fast atoms" support so they will work fine with IRIX releases before IRIX 6.3 and 6.4.

Q30: Can I get a version of GLUT for the Power Macintosh?

A30: Probably pretty soon. Conix Graphics is working on a port of GLUT 3.2 as of late January 1997. Try checking the Conix Graphics web site <u>http://www.conix3d.com/</u> for current info.

Q31: What is new in GLUT 3.4?

A31: GLUT 3.4 is an incremental release. An Ada binding for SGI machines is included along with an Ada example. Many new sample programs. Several such as dinoshade.c demonstrate real-time rendering techniques relevant for games. Examples using Sam Leffler's libtiff library for loading, drawing and writing TIFF image files. GLUT version of the facial animation "geoview" decibed in the Parke and Water's book "Computer Facial Animation". New API interfaces to be made part of the GLUT 4 API update (not yet fully finalized though). glutInitDisplayMode for example. Improved portability and a few bug fixes.

Q32: I installed SGI's Cosmo3D beta and GLUT, and I'm having problems compiling GLUT programs.

A32: Unfortunately, SGI's Cosmo3D beta images install a DSO for GLUT (libglut.so) that does not fully implement the GLUT API and lacks some of the newer GLUT 3.4 entrypoints as well. The problem is that a DSO takes preferenc over an archive when you compile with an option like "-lglut". While the Cosmo3D beta installs a libglut.so, my GLUT distribution and images only build and install an archive. There are a couple of solutions:

- 1. Explicitly link your GLUT programs with libglut.a (the archive version of GLUT). For example, put "/usr/lib/libglut.a" on your compile line instead of "-lglut".
- 2. You can convert the GLUT 3.4 archive into a DSO:

```
su
cd /usr/lib
mv libglut.so libglut.so.cosmo
cc -32 -o libglut.so -shared -all libglut.a
cd /usr/lib32
mv libglut.so libglut.so.cosmo
cc -n32 -o libglut.so -shared -all libglut.a
```

The new DSO generated from the GLUT 3.4 DSO should be compatible with the old Cosmo version. This will mean that all the GLUT programs you build will need the libglut.so on the machine they run on.

3. Remove the Cosmo3D beta.

Q33: What is new in GLUT 3.5?

A33: The most significant change with GLUT 3.5 is unifying the X Window System and Win32 versions of GLUT into a

single source code distribution. Henk Kok contributed several cool new demos (rollercoaster, chess, opengl_logo). All the demos build cleanly under Win32. Lots of bug fixes. Interesting new OpenGL rendering techniques are demonstrated in a number of new examples: movelight, dinoshade, halomagic, rendereps, movelight, shadowfun, torus_test, underwater, texfont, reflectdino.

Q34: How do I use the precompiled Win32 GLUT DLLs with Borland compilers?

A34: The "implib" command should let you generate a GLUT.LIB that works with Borland compilers from the precompiled GLUT.DLL Here is an example:

C:\>implib C:\GLUT\LIB\GLUT.LIB C:\WINDOWS\SYSTEM\GLUT.DLL

After this, then link C:\GLUT\LIB\GLUT.LIB to your project

Suggested by Carter <carter@extremezone.com>.

Q35: Are there any C++ wrappers for GLUT?

A35: Yes, <u>George Stetten (stetten@acpub.duke.edu</u>) of Duke University has made available the GlutMaster C++ wrapper classes. See:

http://www.duke.edu/~stetten/GlutMaster/GlutMaster.html
http://www.duke.edu/~stetten/GlutMaster/README.txt

Q36: How do you avoid the Console window appearing when you compiler a Win32 GLUT application with Microsoft compilers?

A36: Try using the following Microsoft Visual C compiler flags:

/SUBSYSTEM:WINDOWS /ENTRY:mainCRTStartup

These are linker options... if main or wmain are defined, MSVC build a CONSOLE app by default; hence the need for /SUBSYSTEM:WINDOWS. if /SUBSYSTEM:WINDOWS is defined, MSVC expects WinMain or wWinMain to be defined; hence the need to /ENTRY:mainCRTStartup (eg the entry point is the usual C main).

stdout/stderr are [apparently] not "attached"; output via printf is simply "eaten" unless redirected at the command-line or by a parent program.

Information thanks to Jean-David Marrow (jd@riverbed.com).

Q37: What is new in GLUT 3.6?

A37: GLUT 3.6 adds/improves the following:

- Win32 GLUT performance improvements.
- Win32 GLUT confromance improvements.
- <u>Linas Vepstas's GLE Tubing & Extrusions Library</u> is included with GLUT, including nroff man pages and demo programs.
- More GLUT-based OpenGL demos and examples (and bug fixes to existing demos and examples).
- glutPostWindowRedisplay and glutPostWindowOverlayRedisplay entry points added for posting redisplays on non-current windows (for faster multi-window updates).
- Bug fixes and minor functionality improvements to Tom Davis's micro-UI GLUT-based user interface toolkit.

See the "CHANGES" file that accompanies GLUT 3.6 for a fuller list of changes.

Q38: On my IRIX 6.3 SGI O2 workstation, why do I get errors about "glXChannelRectSyncSGIX" being unresolved building certain GLUT examples?

A38: The original IRIX 6.3 release for the O2 workstation accidently advertised support for the dynamic video resize extension supported on SGI's high-end InfiniteReality graphics system. This confuses GLUT into providing its dynamic video resize sub-API.

This problem is fixed by patch 1979 (and its successor patches). Because patch 1979 (and its successor patches) also help O2's OpenGL rendering performance, I strongly recommend requesting the latest O2 OpenGL patch from SGI customer support.

Once the patch is installed, your build errors will be resolved.

Q39: Using GLUT with Microsoft OpenGL 1.1 and compiling GLUT with Borland compilers causes GLUT applications to generates floating point exceptions. What can be done?

A39: Under certain conditions (e.g. while rendering solid surfaces with lighting enabled) MS libraries cause some illegal operations like floating point overflow or division by zero. The default behaviour of Microsoft compilers is to mask (ignore) floating point exceptions, while Borland compilers do not. A function of Borland run-time library allows to mask exceptions. Modify glut_init.c by adding the following lines to the function __glutOpenWin32Connection:

#ifdef __BORLANDC___
#include
 _control87(MCW_EM,MCW_EM);
#endif

With this modification, compiling the GLUT library with your Borland compilers and using GLUT with Microsoft OpenGL should work fine.

GLUT 3.7 will have this change already included in the GLUT library source code distribution.

This advice comes from Pier Giorgio Esposito (mc2172@mclink.it).

Q40: Using GLUT with SGI OpenGL for Windows and compiling with Borland compilers results in linking problems. What can be done?

A40: Some care must be taken when linking GLUT.DLL or programs that use it with Borland compilers. The import library IMPORT32.LIB already contains the functions exported by the Microsoft OpenGL libraries, thus SGI OpenGL import libraries must be listed _before_ import32 in the Borland tlink command line.

This advice comes from Pier Giorgio Esposito (mc2172@mclink.it).

Q41: What is GameGLUT?

A41: GameGLUT is a set of API extension to GLUT to be released in GLUT 3.7. These extensions provide keyboard release callbacks, disabling of keyboard auto repeat, joystick callbacks, and full screen resolution setting.

OpenGL Performance FAQ for NVIDIA GPUs v2.0

John Spitzer

NVIDIA Corporation

JSpitzer@nvidia.com

This document refers to the performance of OpenGL on the NVIDIA GeForce 256, Quadro, GeForce2 MX and GeForce2 GTS, running the Release 5 (5.XX) series of drivers.

I. Geometry

- 1. What are the fastest transfer mechanisms for geometry?
- 2. What are the fastest primitives to use?
- 3. Which vertex array calls should I use?
- 4. How does processor speed and/or bus bandwidth (AGP/AGP2X/AGP4X) affect this?
- 5. What is the best manner in which to organize my geometry in memory?
- 6. What do vertex arrays buy me in terms of performance?
- 7. What do compiled vertex arrays (CVAs) buy me in terms of performance?
- 8. How can I maximize performance with the vertex_array_range extension?
- 9. Is there an optimal size vertex array to define/use?
- 10. Should I use display lists for static geometry?
- 11. Will I get better performance if I chain together separate triangles with degenerate triangle strips?
- 12. I've heard that NVIDIA GPUs have a vertex cache how do I use it?

II. Lighting

- 13. What lighting mode is fastest?
- 14. Which ones should I avoid?
- 15. How many lights should I use?
- 16. Should I turn normalization on or off for maximum performance?
- 17. Should I use the rescale normal extension to increase performance?
- 18. Is it faster if I only want to calculate the diffuse component, not the specular?

III. Texture Coordinate Generation

- 19. Which TexGen modes are hardware accelerated?
- 20. Is the texture matrix hardware accelerated?
- 21. Is there hardware acceleration for two sets of texture coordinates?

IV. Clipping and Culling

- 22. Should I perform any clipping myself?
- 23. Are user-defined clip planes hardware accelerated?
- 24. How many user-defined clip planes are hardware accelerated?

V. Texturing

- 25. How can I maximize texture downloading performance?
- 26. Should I use texture compression?
- 27. How can I maximize texture rendering performance?
- 28. What filtering modes should I use?
- 29. How much performance will anisotropic filtering take away?
- 30. What kind of performance increase can I expect from using paletted textures?

VI. Other Fragment Operations

- 31. What are the performance implications of polygon stippling?
- 32. What fragment operations should I avoid?

VII. Pixel Transfers

- 33. What are the best formats/types to use with glReadPixels and glDrawPixels?
- 34. I want to read back the depth buffer for incremental updates; how should I do this?

VIII. Miscellaneous

- 35. How much will Full Scene Anti-Aliasing (FSAA) slow me down?
- 36. Is context switching expensive?
- 37. What about state changes?
- 38. Why is my GeForce 256 running at a fraction of the speed of my TNT2?
- 39. Should I use a unified back buffer (UBB) or not?

I. Geometry

1. What are the fastest transfer mechanisms for geometry?

Fastest	DrawElements/DrawArrays Using	Saves data in video memory, eliminating any
	wglAllocateMemoryNV(size,0,0,1)	bus bottleneck. Very poor read/write access.
	DrawElements/DrawArrays Using	Saves data in AGP (uncached) memory, and
	wglAllocateMemoryNV(size,0,0,.5)	allows hardware to pull it directly. Very poor
		read access, must write sequentially (see below)
	Display Lists	Can encapsulate data in the most efficient
		manner for hardware, though they are
		immutable (i.e. once created, you can't alter
		them in any way).
	DrawElements using	Copies locked vertices to AGP memory, so that
	Compiled Vertex Arrays	the hardware can then pull it directly. Only one
	(glLockArraysEXT)	mode is supported (see q, 7 below).
	DrawElements and DrawArrays	Optimized to assemble primitives as efficiently
	using Vertex Arrays with Common	as possible, and minimizes function call
	Data Formats	overhead. 13 formats supported (see q. 6).
	Immediate Mode	Multiple function calls required per primitive
		results in relatively poor performance compared
		to other options above.
Slowest	All Other Vertex Arrays	Must be copied from application memory to
		AGP memory before the hardware can pull it.
		Since data can change between calls, data must
		be copied every time, which is expensive.

2. What are the fastest primitives to use?

Fastest	GL_TRIANGLE_STRIP	These maximize reuse of the vertices shared
	GL_TRIANGLE_FAN	within a given graphics primitive, and are all
	GL_QUAD_STRIP	similarly fast.
	GL_TRIANGLES	These aggregate (potentially multiple) disjoint
	GL_QUADS	triangles and quads, and amortize function
		overhead over multiple primitives.
Slowest	GL_POLYGON	A bit slower than the independent triangles and

		quads.
--	--	--------

The GeForce2 GTS is able to setup primitives much faster than GeForce 256 or Quadro, so that all primitives are equally fast when accessing vertices in the vertex cache (see vertex cache question below for other details).

3. Which vertex array calls should I use?

Fastest	glDrawElements	Can take advantage of shared vertices and conserve front-side bus bandwidth by merely sending indices to the data.
	glDrawArrays	The most efficient way to send vertices that are not shared , though much slower than glDrawElements in the common case of shared vertices.
Slowest	glArrayElement	Call overhead per vertex severely impacts performance. Avoid if at all possible.

4. How does processor speed and/or bus bandwidth (AGP/AGP2X/AGP4X) affect this?

Unlike TNT2, NVIDIA GPUs all have hardware T&L, and processor speed is not nearly so important for attaining good T&L performance. Basically, only in immediate mode will processor speed play any significant role in determining T&L performance. Bus bandwidth is another matter, however, as it ultimately limits how quickly the data can pass from system memory to the GPU. AGP4X is needed for optimal performance in many of the transfer modes utilizing the bus, but even in these modes, AGP2X will yield performance close to AGP4X. Standard AGP, on the other hand, creates a bottleneck for many of the transfer modes, and can result in performance far less than optimal.

5. What is the best manner in which to organize my geometry in memory?

There is no inherent advantage, nor disadvantage, to using glInterleavedArrays versus glVertexPointer, glTexCoordPointer, etc. Similarly, there is typically no performance advantage to interleaving data versus keeping the components in separate, disjoint arrays.

6. What do vertex arrays buy me in terms of performance?

Vertex arrays minimize the number of OpenGL calls that must be made to send geometry down the pipeline. Some data formats are specifically optimized for use within regular vertex arrays:

Vertex Size/Type	Normal Type	Color Size/Type	Texture Unit 0 Size/Type	Texture Unit 1 Size/Type	Fog Coord Type
3/GLfloat	-	-	-	-	-
3/GLfloat	-	-	2/GLfloat	-	-
3/GLfloat	-	-	2/GLfloat	2/GLfloat	-
3/GLfloat	GLfloat	-	-	-	-
3/GLfloat	GLfloat	-	2/GLfloat	-	-
3/GLfloat	GLfloat	=	2/GLfloat	2/GLfloat	-
3/GLfloat	-	3/GLfloat	-	-	-
3/GLfloat	-	3/GLfloat	2/GLfloat	-	-
3/GLfloat	-	3/GLfloat	2/GLfloat	2/GLfloat	-

3/GLfloat	-	4/GLubyte	-	-	-
3/GLfloat	-	4/GLubyte	2/GLfloat	-	-
3/GLfloat	-	4/GLubyte	2/GLfloat	2/GLfloat	-
3/GLfloat	-	-	2/GLfloat	3/GLfloat	-
3/GLfloat	-	-	-	-	GLfloat
3/GLfloat	-	-	2/GLfloat	-	GLfloat
3/GLfloat	-	-	2/GLfloat	2/GLfloat	GLfloat

Other formats are likely to be slower than immediate mode.

7. What do compiled vertex arrays (CVAs) buy me in terms of performance?

Although your mileage may vary, compiled vertex arrays can yield a large increase in performance over other modes of transport – specifically, if you frequently reuse vertices within a vertex array, have the appropriate arrays enabled and use glDrawElements. Only one data format is specifically optimized for use within CVAs:

Vertex	Normal Type	Color	Texture Unit	Texture Unit
Size/Type		Size/Type	0 Size/Type	1 Size/Type
3/GLfloat	-	4/GLubyte	2/GLfloat	2/GLfloat

Note that there is no corresponding glInterleavedArrays enumerant for this format (i.e. you must use glVertexPointer, glColorPointer and glTexCoordPointer to specify the arrays).

When using compiled vertex arrays with this format, it's important to maximize use of the vertices that have been locked. For example, if you lock down 100 vertices and only use 25 of them in subsequent glDrawElements calls before unlocking, you will have relatively poor performance.

For more flexibility in accelerated data formats, it's recommended that vertex_array_range extension be used (see below).

8. How can I maximize performance with the vertex_array_range extension	1?
Currently, you should only use the vertex_array_range with memory allocated by	
wglAllocateMemoryNV (or glXAllocateMemoryNV) given the following settings	:

"gir moedlentemory" (or girl moedlentemory" () given the fonowing setting					
Memory Allocated	ReadFrequency	WriteFrequency	Priority		
AGP Memory	[0, .25)	[0, .25)	(.25, .75]		
Video Memory	[0, .25)	[0, .25)	(.75, 1]		

All other settings will yield relatively poor performance. Use video memory sparingly, and only for static geometry. You may use AGP memory for dynamic geometry, but write your data to these buffers sequentially to maximize memory bandwidth (it is uncached memory, and sequentially writing is **essential** to take advantage of the write combiners within the CPU that batch up multiple writes into a single, efficient block write). And being uncached, read access will be very, very slow – it may be best to keep two buffers, one allocated by standard malloc for general R/W access and the other allocated by wglAllocateMemoryNV that is only written to – synchronization would copy data from the R/W buffer sequentially into the AGP memory. Keep the vertex array strides to a reasonable length (less than 256), and mind the necessary alignment restrictions in the extension specification. Also, do not use wglAllocateMemoryNV unless you

use and enable the vertex_array_range extension. If you do not heed those restrictions, you will certainly have less than optimal (if not poor) performance.

9. Is there an optimal size vertex array to define/use?

There is no hard and fast rule for vertex array size with respect to performance. Allocating a huge amount of AGP memory is probably not wise, however, since that memory will not be available to the OS (possibly causing unnecessary thrashing).

10. Should I use display lists for static geometry?

Yes, they are simple to use and the driver will choose the optimal way to transfer the data to the GPU.

11. Will I get better performance if I chain together separate triangles with degenerate triangle strips?

No. Draw what you can with triangle strips (or quad strips) and triangle fans, then draw the remaining independent triangles using glBegin(GL_TRIANGLES)...glEnd().

12. I've heard that NVIDIA GPUs have a vertex cache – how do I use it?

All NVIDIA GPUs have a 16 element post-T&L vertex cache (also called the "vertex file"), though the effective size is closer to 10 elements when you consider pipelining. You must adhere to a few rules to take advantage of the vertex cache:

- 1. Use glDrawElements or glDrawRangeElements
- 2. Use the NV_vertex_array_range extension (see question 8 above) or the optimized compiled vertex array format (see question 7 above).
- 3. Ensure your vertices are shared between multiple primitives, and that you have decent vertex cache coherency (i.e. adjacent triangles are drawn together)

II. Lighting

13. What lighting mode is fastest?

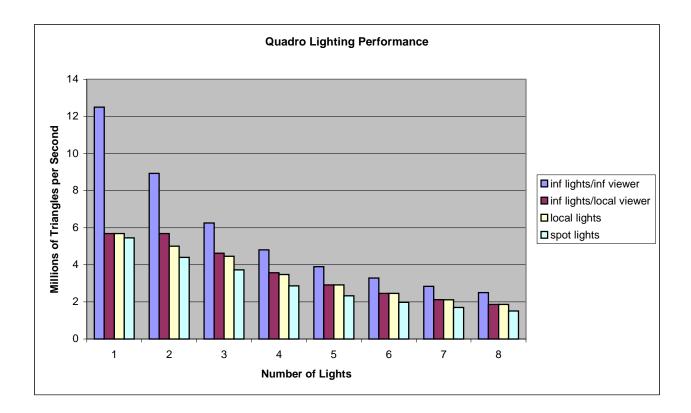
Directional (AKA infinite) lights, with infinite (i.e. non-local) viewer and one-sided lighting.

14. Which ones should I avoid?

When using directional lights, avoid using local viewer, because it may cut your performance in half. However, when using local lights, you can use local viewer "for free" – that is, the GPU can calculate the local viewer at the same performance as infinite viewer. Two-sided lighting will be slower than one-sided lighting, and should only be used when absolutely necessary.

15. How many lights should I use?

In general, use as few as possible. When using local lights with attenuation, far-off lights will often not contribute to a given surface, although many calculations will still have to be made by the GPU. You can optimize for this by reducing the number of enabled lights to those in an object's immediate vicinity. Reference the graph below to see how much additional lights cost.



16. Should I turn normalization on or off for maximum performance?

The GPU performs normalization very efficiently, so that the cost for enabling it is negligible. Since unexpectedly bright or dim lighting can occur if normalization is disabled (with non-unit length normals), it's recommended that you always enable normalization.

17. Should I use the rescale normal extension to increase performance?

No, enable normalization instead. [See question above]

18. Is it faster if I only want to calculate the diffuse component, not the specular?

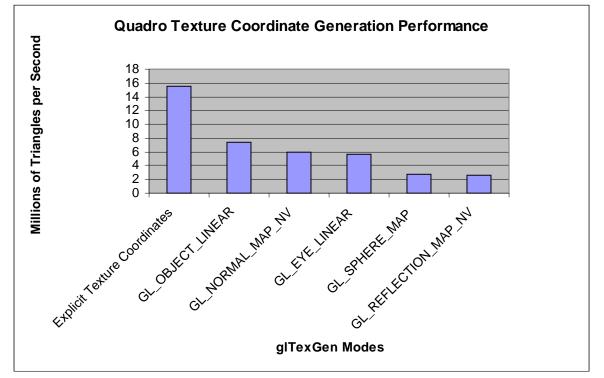
If you want to only compute the diffuse component – presumably by setting the specular material to black – additional performance will not be gained on NVIDIA GPUs, which include the specular calculation "for free". Separating the diffuse and specular colors and applying the specular component after texturing also incurs no additional T&L cost, though rasterization of large, non-Z buffered polygons may be slower.

Specifically, GeForce2 GTS interpolates the secondary/specular color at full speed. GeForce2 MX, GeForce 256 and Quadro all check if the specular color is constant across all the vertices of a triangle. If constant, the interpolation unit runs at full speed because there is no need to interpolate the secondary color. However, if the color varies over the triangle, the color interpolators have to be double pumped with the diffuse and specular color, which will cause that unit to run at half speed. Bear in mind that this will reduce overall performance only if this unit is already the bottleneck.

II. Texture Coordinate Generation

19. Which TexGen modes are hardware accelerated?

All 6 TexGen modes are hardware accelerated, but not at similar performance. See chart below.



20. Is the texture matrix hardware accelerated?

Yes, transformations for both texture units are performed in the GPU. There may be a performance penalty associated with the texture matrix. While the maximum performance on Quadro with an identity texture matrix is almost 16M triangles/second (see graph above), the performance will drop to around 10 M triangles/second with a non-identity texture matrix. If you are transfer-bound or raster-bound, you will not see any performance drop at all.

21. Is there hardware acceleration for two sets of texture coordinates?

Yes. This includes two TexGen units and two texture matrices.

III. Clipping and Culling

22. Should I perform any clipping myself?

No, it's fastest to allow OpenGL to handle it, since the GPU performs viewport clipping very efficiently. In order to take advantage of this clipping, applications should pass in unclipped geometry. Applications should continue to perform gross culling against the view frustum before sending complex objects, and some intelligent scene occlusion culling, such as a BSP.

23. Are user-defined clip planes hardware accelerated?

Yes, a number of user-defined clip planes are hardware accelerated through use of texture mapping and special hardware features.

24. How many user-defined clip planes are hardware accelerated?

For every texture unit you have left unused, you get two hardware user-defined clip planes. The caveat is that enabling polygon stipple counts as using a texture unit if you are not already using both texture units (see question on polygon stippling below). For example, you can use 2 clip planes with single-texturing, and 4 clip planes with no texturing, assuming no polygon stipple.

If more clip planes are defined than can be implemented by the hardware, the driver falls back to software clipping. If lighting is disabled, the driver can use fairly fast clip routines. However, clip planes are harder when lighting is enabled, because you have to light the vertices and then apply the clip planes, interpolating the lighted vertex results to the clipped coordinates. If lighting is enabled, the driver must use fairly slow clipping code. Avoid this case, if at all possible. In fact, avoid user-defined clipping planes altogether, if possible.

IV. Texturing

25. How can I maximize texture downloading performance?

Best RGB/RGBA texture image formats/types in order of performance:

Image	Image Type	Texture Internal
Format		Format
GL_RGB	GL_UNSIGNED_SHORT_5_6_5	GL_RGB
GL_BGRA	GL_UNSIGNED_SHORT_1_5_5_REV	GL_RGBA
GL_BGRA	GL_UNSIGNED_SHORT_4_4_4_REV	GL_RGBA
GL_BGRA	GL_UNSIGNED_INT_8_8_8_REV	GL_RGBA
GL_RGBA	GL_UNSIGNED_INT_8_8_8_8	GL_RGBA

Bear in mind that the NVIDIA GPUs store all 24-bit texels in 32-bit entries, so try using the spare alpha channel for something worthwhile, or it will just be wasted space. Moreover, 32-bit texels can be downloaded at twice the speed of 24-bit texels. Single or dual component texture formats such as GL_LUMINANCE, GL_ALPHA and GL_LUMINANCE_ALPHA are also very effective, as well as space efficient, particularly when they are blended with a constant color (e.g. grass, sky, etc.). Most importantly, **always** use glTexSubImage2D instead of glTexImage2D (and glCopyTexSubImage2D instead of glCopyTexImage2D) when updating texture images. The former call avoids any memory freeing or allocation, while the latter call may be required to reallocate its texture buffer for the newly defined texture.

26. Should I use texture compression?

If image fidelity is not of utmost importance, you should definitely consider using texture compression via the GL_ARB_texture_compression (<u>http://oss.sgi.com/projects/ogl-sample/registry/ARB/texture_compression.txt</u>) and GL_texture_compression_s3tc (<u>http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt</u>) extensions. This technology allows larger textures to be put in a smaller space, and thus reduces the chance of texture thrashing. The compressed texture can also be downloaded much faster to the GPU. See the NVIDIA developer web site for a white paper and code example.

27. How can I maximize texture rendering performance?

In general, it's best to minimize the number of texture binds that must be performed. By sorting its objects by texture, an application can optimally render them in order. Moreover, try to eliminate multiple passes over the same object by using multitexture. Although performing 2 texture multitexture may be more expensive than a single texture on GeForce 256 and Quadro, it is still **much** cheaper than performing another pass. And under most conditions, GeForce 2 GTS can perform dual-texturing at the same rate as single-texturing. Also consider using the GL_NV_register_combiners extension to reduce the number of required passes. This extension provides a good deal of flexibility in combining RGB and ALPHA components, as well as exposing functions such as dot products on a per-pixel level. If using register combiners, try to use only one general combiner, since using two combiners may lower texturing performance. By all means, however, use two general combiners rather than create another rendering pass!

28. What filtering modes should I use?

Mipmapping is always advised, particularly for minified objects. Minified objects create a very large stride through the non-mipmapped texture image, yielding poor cache utilization. Mipmapping greatly reduces the stride and results in higher cache utilization, and in turn, higher performance. Choose the 4-tap GL_LINEAR_MIPMAP_NEAREST as a minification filter, as it will be faster than the 8-tap trilinear filter.

29. How much performance will anisotropic filtering take away?

Anisotropic filtering comes at a slight performance penalty on NVIDIA GPUs (around 10%). It is most effectively used in a "Highest Quality" mode in concert with trilinear texture filtering.

30. What kind of performance increase can I expect from using paletted textures?

Paletted textures will not create a large increase in performance, and will possibly even decrease performance if shared palettes are not employed.

VI. Other Fragment Operations

31. What are the performance implications of polygon stippling?

Polygon stippling is implemented via texture mapping in NVIDIA GPUs, and though fast, it burns a texture unit. Performing polygon stippling with dual-texturing will force the driver to render in software.

32. What fragment operations should I avoid?

Try to curb use of blending, because it requires a read/modify/write operation. All non-zero blending modes cut fillrates in half (compared to non-blended rendering). Use alpha test instead of blending where feasible (e.g. to render sprites and such). One operation to stay away from is the color logical operator (as known as logic op). Only the GL_COPY operator is hardware accelerated on NVIDIA GPUs, relegating the rest to software rendering, at a mere fraction of the hardware's performance.

VII. Pixel Transfers

33. What are the best formats/types to use with glReadPixels and glDrawPixels?

For 32-bit glReadPixels, stick to using GL_UNSIGNED_BYTE type, with GL_RGB, GL_RGBA and GL_BGRA formats. For 16-bit glReadPixels, use GL_FLOAT type, with GL_RGB format. For 16/32-bit glDrawPixels, use GL_UNSIGNED_BYTE type, with GL_RGB and GL_RGBA formats. A type of GL_FLOAT will also give decent, though lower, performance with these formats.

34. I want to read back the depth buffer for incremental updates; how should I do this?

Writing to the depth buffer via glDrawPixels is quite slow (though reading the depth buffer via glReadPixels is moderately fast). For performing incremental updates to scenes by saving away the color and depth buffers, consider using the GL_KTX_buffer_region extension.

VIII. Miscellaneous

35. How much will Full Scene Anti-Aliasing (FSAA) slow me down?

Your mileage will vary depending upon which part of the system is the performance bottleneck. In general, if you are limited by anything but rasterization (e.g. your CPU's speed, or T&L performance), FSAA should not incur any cost at all. If you're limited by rasterization, however, your performance will drop in proportion to your super-sampling rate. For example, 2X FSAA requires just over two times the rasterization as no-FSAA, while 4X FSAA requires four times the rasterization as no-FSAA.

FSAA causes more video memory to be used, so texture thrashing may occur with FSAA enabled, where it did not occur with it disabled.

36. Is context switching expensive?

Yes. Context switching is often a problem in workstation applications, though not as commonly a problem in games. Reduce context switching to a minimum by reusing a single context, and bind it to separate windows, if necessary. It's best to have merely a single window/context and use glViewport and glScissor to restrict rendering to specific "sub-windows".

37. What about state changes?

State changes can severely impact performance. As such, they should be minimized by binning primitives with similar state (textures first, then lights, then blending modes, materials and so on) and drawing them all at the same time.

38. Why is my GeForce 256 running at a fraction of the speed of my TNT2?

Chances are, you have antialiased polygons enabled (i.e. glEnable(GL_POLYGON_SMOOTH)) and you're running on 3.XX drivers. If you turn it off, performance will increase dramatically.

39. Should I use a unified back buffer (UBB) or not?

Quadro has the ability to enable a unified back buffer (in fact, it's enabled by default). The unified back buffer is particularly useful for applications that use many (overlapping) windows and cannot afford to create separate back-buffers for each of them, since it uses too much framebuffer memory. UBB may be slightly slower for single windowed apps, so if you're running games, it's usually not a good idea to have it enabled. If you're running workstation apps, however, it probably **is** a good idea to enable it.

Mesa Frequently Asked Questions

Introduction

For general OpenGL questions see the OpenGL FAQ

Most questions regarding Mesa can be answered by reading the README files included with Mesa. Please do so before sending email questions.

Suggestions for topics to add to this document are welcome.

Compilation and installation problems

Mesa doesn't compile on my system

First, make sure you have the latest version of Mesa. If a newer version doesn't help, keep reading.

Can't compile on Windows 95/NT using (any) compiler

First, I (Brian) do not develop on 95/NT so I can't help you. Look in the Mesa README files for people to contact for help. Otherwise, ask on the Mesa mailing list.

I compiled my program with Mesa but the linker reports all kinds of undefined symbols such as bgnpolygon, v3f, etc...

The program was written for IRIS GL, not OpenGL (which Mesa emulates). Perhaps you should convert your application to OpenGL.

I'm trying to compile Mesa for 'linux-elf' but get lots of errors.

If the errors look like this:

```
/usr/lib/crt0.o(.text+0x35): undefined reference to `main'
accum.o(.text+0x1d): undefined reference to `GLOBAL_OFFSET_TABLE_'
accum.o(.text+0x224): undefined reference to `GLOBAL_OFFSET_TABLE_'
etc...
```

Then you probably don't have a gcc compiler with ELF support even though your kernel can run ELF binaries. Try gcc-2.7.2 or later.

Can't compile Mesa 2.2 (or ealier) on Redhat 4.1

It appears they've changed the X11 development directories a bit. Try adding -I/usr/X11R6/include to the CFLAGS for linux in the Make -config file. You may also have to add -L/usr/X11R6/lib to the XLIBS line.

Can't compile Mesa 2.x on RedHat 5.0

Get Mesa 2.6.

Runtime problems

Mesa FAQ

I'm seeing errors in depth (Z) buffering

Make sure the ratio of the far to near clipping planes isn't too great. Look here for details.

If that doesn't help then edit the src/config.h file and change DEPTH_BITS to 32 instead of 16. Mesa uses a 16bit depth buffer by default which is smaller and faster to clear than a 32-bit buffer but not as accurate.

Depth buffering isn't working at all

Be sure you're requesting a depth buffered-visual. If you set the MESA_DEBUG environment variable it will warn you about trying to enable depth testing when you don't have a depth buffer.

glGetString() always returns NULL

Be sure you have an active GL context before calling glGetString.

I've compiled Mesa on my Linux (or other Unix system) but I get not DGL-capable errors.

There are two problems here. First, your application uses IRIS GL, not OpenGL. Find and OpenGL version of the application or rewrite it to use OpenGL.

Second, if you want to remotely display OpenGL apps from your SGI onto another X server you must compile and install Mesa on the SGI, not on your local host. See below for more information about GLX.

Mesa crashes on my Linux PC with "signal 11" or other errors when using the *linux-386* configs

Try upgrading the *binutils* on your system. Older versions of binutils don't correctly assemble the 386 code in Mesa.

Points, lines, triangles aren't rendered on the exact pixel I expect

This is probably a problem involving point sampling and numerical round-off error. See the appendix of the OpenGL Programming Guide for the solution.

Hardware support

Is anybody working on supporting 3-D PC hardware?

Yes, for currently available hardware support see the info at the bottom of the main Mesa page.

There's at least one or two other Linux 3-D hardware projects underway that haven't been officially announced yet. I can's say any more.

What about S3 Virge support with Linux?

Someone at S3 is working on this in his spare time. There is no estimated time for release.

What about the nVidia Riva128?

There are no plans to support it at this time. nVidia hasn't released their hardware specs to the public. Perhaps if a group of people would organize themselves and contact nVidia they could get technical specs under NDA. You should post to the Mesa mailing list if interested in doing this.

What about support for the XYZ Inc. SuperTurboMega3D card?

First, a 3-D card cannot be supported with Mesa unless complete technical specifications are available for the hardware. Most vendors don't release this info, except perhaps under NDA. Second, someone has to volunteer to

Mesa FAQ

write the Mesa driver for the 3-D hardware.

I (Brian) do not have the time to do this myself. However, I'm happy to assist anyone working on new Mesa drivers.

Ultimately, 3-D hardware acceleration for Linux should be integrated into the (XFree86) X server. If you're interested in doing the work you should contact the XFree86 development group.

Will Mesa work with Voodoo2 (on Linux or Windows)?

Voodoo2 requires that the 3Dfx Glide library be updated. After that has been done Mesa will work with Voodoo2.

Miscellaneous

Can I use Mesa with Ousterhout's Tcl/Tk?

Yes, check out Togl

Can I use Mesa to display OpenGL applications on my X terminal?

Yes. If you have source code to the application, just recompile and relink with Mesa instead of OpenGL.

If you do not have souce code to the application you can still use Mesa if your operating system supports dynamic run-time linking (such as IRIX 5.x). Basically, you have to create a Mesa shared library (DSO under IRIX 5.x) and tell the runtime loader/linker to use the Mesa library instead of the OpenGL library. Usually this is done by creating a symbolic link to the Mesa library which is the same as the OpenGL library name. Then set a shell environment variable (_RLD_LIST on IRIX 5.x) to point to the symbolic link. An example of this for IRIX 5.x is in the Mesa README file.

I compiled Mesa on my Linux (or other) computer but when I try to view programs on my PC which are running on my SGI I get "GLX missing" errors

Compiling Mesa on your Linux PC <u>does not</u> install the GLX extension into your X server. You have to compile Mesa on the SGI as explained in the previous step.

Is Mesa multithreaded?

Thread support in Mesa is under development. Contact John Stone (johns@umr.edu) for details.

Linux Quake questions

A lot of people use Mesa on Linux to run Quake...

If you have problems, be sure you have the latest version of Mesa. Then, perhaps ask for help on the <u>3dfx.glide.linux</u> newsgroup.

Last updated on February 23, 1999.

Mesa User's Guide

Introduction

This document discusses a number of Mesa subjects which are often asked about. Also see the FAQ for miscellaneous subjects. Suggestions for new topics are welcome.

Subjects:

- 1. X visuals and colormaps
- 2. Mesa's X driver
- 3. Optimizing Mesa's performance
- Installing Mesa
 Remote display of OpenGL apps

Subject 1: X visuals and colormaps

Before explaining the details of the X driver for Mesa (subject 2) it's important to understand some basic information about X visuals. Volume 1 of the O'Reilly series on the X window system is a good source for this information.

Visuals

An X "visual" describes how data in a frame buffer are displayed as colored pixels on your screen. Visuals are characterized by their depth and class. The depth is the number of bits per pixel. The class determines what kind of colormap, if any, is used.

X supports 6 different visual types or classes (N=depth):

- StaticGray each N-bit pixel values is an index into an immutable grayscale colormap with 2^N entries
- GrayScale each N-bit pixel value is an index into a mutable grayscale colormap with 2^N entries
- StaticColor each N-bit pixel value is an index into an immutable colormap with 2^N entries
- **PseudoColor** each N-bit pixel value is an index into a mutable colormap with 2^N entries
- **TrueColor** each N-bit pixel values is partitioned into 3 components (R + G + B = N) which directly map to 2^R red, 2^G green, and 2^B blue intensities
- **DirectColor** each N-bit pixel value is partitioned into 3 components (R + G + B = N) each of which is an index into a mutable red, green, and blue colormap

Note: mutable = dynamic or changable, immutable = fixed, can't be changed.

The most common visual type on low-end displays is 8-bit PseudoColor. In this case each byte in the frame buffer is an index into a 256-entry colormap which can be loaded with colors you choose.

A common visual type on high-end displays is 24-bit TrueColor. In this case each triplet of bytes in the frame buffer directly maps to an RGB color on the screen. 256 shades of red, 256 shades of green and 256 shades of blue allow 16,777,216 differeent colors. Some people say you can display "16 million colors at once" but that's false because nobody has a display with that many pixels!

Here are some other common visuals:

- 1-bit StaticGray monochrome screen
- 8-bit GrayScale grayscale screen
- 8-bit TrueColor 2-bits red, 3-bits green, 3-bits blue
- 12-bit PseudoColor 12 bits per pixel, 4096-entry colormap
- 16-bit TrueColor 5-bits red, 6-bits green, 5-bits blue

Which visual(s) does my display support?

You can find out with the standard **xdpyinfo** command. It prints all sorts of interesting information about your display including a list of visuals supported by each screen. Note that an X display is a collection of one or more X screens, each of which can support a different set of visuals. Most people have one screen per display. Low-end systems usually list 1 or 2 visuals, high-end systems may list upwards of 70 visuals.

Which visual is the default?

One of the visuals in the list from xdpyinfo is the default visual. The default visual is the visual used by the root (background) window. Look for *default visual id* in the xdpyinfo output.

Another way to determine the default (root) visual is to use **xwininfo**. When you run xwininfo your pointer will turn into a cross-hair. Point over the root window and press a mouse button. Among the information printed will be the visual class and depth. Note that you can apply this program to any X window.

Can I control which visuals are available?

That depends on your graphics hardware and X server software. On Linux systems with XFree86 you can do startx -- -bpp16 or startx -- -bpp32 to start the X server with deeper visuals. Ask your systemin or consult your system's X documentation to learn more.

If your display supports more than one visual you should also be able to configure the default (root) visual to be which ever you want. Again, read your documentation.

Information for Xlib programmers

If you're programming with Xlib (or a higher level toolkit) you need to be aware of visual issues when creating windows. Naive programmers who use XCreateSimpleWindow may find all kinds of problems when later running their client on a different display. The problem is XCreateSimpleWindow inherits its parent's visual. If creating a top-level window, it'll inherit the root win window's visual which will vary from display to display.

When creating top-level windows it's much better to use **XGetVisualInfo** or **XMatchVisualInfo** to explicitly choose the visual and **XCreateWindow** to create the window. Alternatively, if you want to use the default visual, your code should verify that the default visual is suitable for your application's needs and deal with it appropriately.

Finally, If you create a window with a visual you've explicitly chosen you must also be sure to provide a colormap which matches the visual. Otherwise you'll get a BAD MATCH X protocol error.

Colormaps

Color management in X is complicated. What follows is a quick overview of X's colormap system. See the O'Reilly Xlib Programming Manual for more detailed info.

An X colormap is really an abstraction over the hardware. While your X screen may only have one real colormap, X gives programmers the illusion of having an unlimited number of colormaps. If the hardware colormap(s) become over commited you'll probably see the "technicolor" effect or colormap "flashing" when you move the input focus from one window to another. That's caused by the window manager installing the virtual X colormap into the hardware colormap for the current window. Careful programming can reduce or eliminate this problem as we'll see.

X colormaps come into two varieties: private and shared. When you call **XCreateColormap** you indicate **AllocAll** for private or **AllocNone** for shared.

When you create a private colormap you get a whole colormap to yourself in which you can setup any mapping of pixels to colors you want using **XStoreColor(s)**. You should avoid using private colormaps when possible because they inhibit color sharing. Remember, it's not sharing colors with other clients which leads to the dreaded colormap flashing.

When you create a shared colormap you must allocate colors from it using **XAllocColor**. You specify a color by red, green, and blue values and XAllocColor returns a pixel value for you to use when drawing things. If X can't allocate the color you need, XAllocColor will fail. Your best recourse is to then search the colormap for the closest match and use that color. X will try to combine shared colormaps into one hardware colormap to reduce flashing.

Programming tips:

- Use shared colormaps whenever possible.
- If you need to create a number of windows with the same visual you should try to share the same colormap among them.
- If you create a number of windows with different visuals you *must* be sure to allocate a different colormap for each visual.

• If possible, try to use the visual and colormap of the root window to reduce colormap flashing. How Mesa works with colormaps is the subject of the next section.

Subject 2: Mesa's X driver- visuals and colormaps

According to the OpenGL GLX spec, when using OpenGL in RGB mode you must use a TrueColor or DirectColor visual. When using OpenGL in color-index mode you must use a PseudoColor or StaticColor visual. Indeed these are the only possibilities returned by **glXChooseVisual**.

Mesa's X driver is more flexible, allowing you to use any X visual type in RGB mode and either GrayScale, StaticGray, PseudoColor or StaticColor in color- index mode. Unfortunately, this flexibility sometimes causes problems.

It's very important to understand that most of the visual and colormap problems people have with Mesa are not caused by the core Mesa library but rather the higher level toolkits such as aux, tk and GLUT. However, the toolkits cannot be blamed too much because they were designed to work with OpenGL but not Mesa's unique features.

Mesa's glXChooseVisual

Mesa's implementation of **glXChooseVisual** is written to be as compatible with the OpenGL semantics as possible. However, The fact that Mesa's glXChooseVisual may return, for example, a PseudoColor visual in RGB mode is enough to make some OpenGL applications fail. If the OpenGL application requires a TrueColor or DirectColor visual and your display doesn't support such a visual you may be out of luck. This is no one's fault. However, if you write an OpenGL application, you'd be doing a service to Mesa users if you wrote code which would accept any visual type in RGB mode.

Remember that if Mesa's glXChooseVisual were modified to behave exactly like OpenGL's we would actually be losing functionality which a lot of people (everyone without a TrueColor display) depend on.

How can I stop colormap flashing?

If the colors on your screen flash when you move the pointer in and out of a Mesa window it's because the working set of Mesa and other X clients have allocated more colors than will fit in the hardware colormap(s). To remedy this, you can either close some of your other X clients or try setting the MESA_RGB_VISUAL environment variable to match the root window's visual, thereby encouraging colormap sharing.

I don't see flashing but the Mesa window's colors are wrong!

Your Mesa window is probably using the same visual type as the root window and is sharing the root's colormap. Unfortunately, either the window manager and/or other X clients have allocated so many entries from the colormap that Mesa can't get the ones it needs for its palette. The solution is to try the Mesa application again after you've terminated other color-demanding clients. Or set the MESA_PRIVATE_CMAP variable which forces the aux, tk and GLUT toolkits to allocate a private colormap. Unfortunately, now you'll probably see colormap flashing.

Note that the MESA_PRIVATE_CMAP variable is recognized by the aux and tk toolkits and **not** the Mesa core library. Colormap management is an issue at a level above the core of Mesa.

Caveat

The above discussion assumed you're using Mesa in RGB mode. If you're using color-index mode most of the above is still applicable. However, many (most?) color-index mode application need a private colormap so they can manipulate (read/write) the colormap. If, for example, your display does not have a PseudoColor visual the Mesa/OpenGL application many generate X protocol errors when it tries to execute an XStoreColor command.

Subject 3: Optimizing Mesa's performance

The following is a list of things you can do to maximize the performance of Mesa. In no particular order...

Experiment with the **MESA_BACK_BUFFER** environment variable if using double buffered mode. Possible values are "P" for pixmap and "X" for XImage. When displaying on the local host and using an XImage for the back buffer, the X shared memory extension is used to accelerate the glXSwapBuffers() function. Using an X image is usually faster except when rendering scenes which don't use any raster operations (such as depth-test, stenciling, dithering, etc) since the Xlib point, line and polygon functions can be used.

Experiment with different **visuals** with the MESA_RGB_VISUAL environment variable. Some are visuals faster than others.

Try to maximize the number of vertices between glBegin/glEnd.

Group state changes such as glEn/Disable, glShadeModel, etc together before glBegin/glEnd to minimize the number internal state change computations.

Disable smooth shading when not needed. Smooth shading is usually only needed for drawing lit polygons.

Disable dithering when not needed.

Disable depth testing and any other raster operations you don't need.

glDrawPixels works quickest with GL_UNSIGNED_BYTE, GL_RGB - format image data.

Use GLfloat-valued functions such as glVertex f[v], glNormal3f[v], glColorf[v] glLoadMatrixf, glMultMatrixf, etc. because conversion to the internal GLfloat type will not be needed.

Use backface culling to reduce the rasterization bottleneck.

Using a smaller window will speed up polygon rasterization, glClear, and glXSwapBuffers.

Avoid using **glColorMaterial**.

Use **directional lights** rather than positional lights. i.e. W component of position = 0.0.

Avoid using **GL_LIGHT_MODEL_LOCAL_VIEWER**.

Avoid using spot lights.

Use **low-numbered**, **consecutive lights** such as GL_LIGHT0, GL_LIGHT1, GL_LIGHT2 rather than GL_LIGHT2, GL_LIGHT5, GL_LIGHT7 for example.

Avoid using GL_NORMALIZE.

Use **viewports** which are completely inside the window boundaries.

Subject 4: Installing Mesa (on Unix systems)

After you've compiled the Mesa library files, as seen in Mesa/lib, you should probably move them and the include files to a more appropriate location. I suggest copying the Mesa/lib files to /usr/local/lib and copying the Mesa/include/GL directory to /usr/local/include.

When you compile your Mesa/OpenGL application just add -I/usr/local/include to your C compiler flags and add -L/usr/local/lib to your linker flags.

If your system doesn't have real OpenGL libraries it may also be a good idea to make a few symbolic links so that

"off the shelf" OpenGL applications compile painlessly:

```
ln -s /usr/local/include/GL /usr/include/GL
ln -s /usr/local/lib/libMesaGL.a /usr/lib/libGL.a
ln -s /usr/local/lib/libMesaGLU.a /usr/lib/libGLU.a
```

NOTE: if you've made shared Mesa libraries the symbolic links will probably have different names: .so suffix instead of .a suffix, for example. If you do this you may also have to run a special program such as ldconfig - v on Linux to make things work.

Then you can specify $_{-1GL}$ and $_{-1GLU}$ when linking your Mesa application and be confident that it will also compile successfully on other systems which may have real OpenGL libraries.

Subject 5: Remote display of OpenGL apps

Normally, X11-based OpenGL applications can only be displayed on X servers which have the GLX extension. The GLX extension decodes the GLX protocol (which is sent within the X protocol stream) and executes the appropriate OpenGL rendering operations. You can check if your display server has this extension by examining the output of running xdpyinfo.

If you have an OpenGL application and want to display it on a server which lacks the GLX extension, Mesa can help you. You have two alternatives:

- If you have the application source, recompile it (or just relink it) using the Mesa libraries instead of the OpenGL libraries. Basically, just substitute -lGL with -lMesaGL in the Makefile. The application should now be displayable on almost any X server.
- 2. If you don't have the application source *but* it was linked with a *shared* OpenGL library you can replace the OpenGL shared library with the Mesa shared library at runtime. Naturally, this requires that your operating system uses shared libraries (i.e. IRIX, Linux 1.2.x, SunOS, AIX, HPUX and others).

If you're not familiar with shared libraries you should read your system's documentation. Man pages on ld, rld, ld.so or man -k library should turn up something.

Here are the steps to using a Mesa shared library in place of OpenGL:

- 1. You have to compile Mesa as a shared library. The Mesa Makefile already supports this for a number of systems. Just type make in the Mesa directory to see a list of configurations and look for yours.
- 2. Make a symbolic link with the same name as your system's OpenGL library which points to the Mesa library. For example, on IRIX systems the OpenGL lib is named libGL.so so you'd create the symbolic link with: ln -s libMesaGL.so libGL.so in the Mesa/lib directory. Note that you could just rename the Mesa library instead of making a symbolic link, if you prefer.
- 3. Tell the runtime linker to look in Mesa/lib (or where ever you've installed the Mesa shared library) for libraries before the default library directories. On IRIX 5.x systems this is done by setting the __RLD_LIST environment variable: setenv _RLD_LIST "mesalibdir/libGL.so:DEFAULT" where mesalibdir is the full path to the location of the symbolic link you made previously.

Now when you execute the OpenGL application the runtime linker should select the Mesa shared library instead of the OpenGL shared library.

Using either of these methods, The application should now be displayable on almost any X server since the OpenGL API calls will effectively be translated into ordinary X protocol by Mesa.

Why did I say "almost any X server"? Because it might be the case that the OpenGL application won't accept any of the visual types offered by your display. For example, if the OpenGL app asks for an RGBA visual and Mesa returns a PseudoColor visual the application may not accept it because a TrueColor or DirectColor visual was expected. You may have to experiment with the MESA_RGB_VISUAL environment variable if you have this problem.

Back to the <u>Mesa home page</u>

Last updated on January 19, 1996 by brianp@ssec.wisc.edu.



library.

About

News

Home

Download

A full featured cross-platform image

.OIL Specifications Last Revised: 5:55 AM 12/24/2000

The .oil file format was developed to be a robust solution to the lack of truecolour animated images (.mng is a possible one, but I've never even seen a .mng file, the library is still in beta and the format lacks some desirable features). From this auspicious beginning, .oil blossomed into a full-fledged image format, designed to support future SourceForge additions (such as new types of compression) without breaking earlier files.

Or, as Aggrav8d of #flipCode said: **Documentation** Say something like, "It was conceived by a comity formed by the clones of history's greatest minds, inscribed on sheets of silk by immaculate virgins using ink made by **Tutorials** blind monks who used ground down charred pieces of the true cross. It was prompty lost in a sea of paganistic anarchy for a thousand years, kept secret by templar knights Logos and guarded by the last great Chinese dragon and curses more powerful than those of Tutankhamen until DooMWiz performed an ancient series of rituals and rights-of-Links passage until finally he was allowed the right to glimpse it's wonderous magnificence. He promptly stole it." Projects

Contact Us .oil files are always in little endian format.

File Header

The .oil format is powerful, yet easy to read and begins with the obligatory file header.

ypedef struct OILHEAD
char ByteHead[4];
ILuint MagicNum;
ILushort Version;
ILuint NumImages;
ILuint DirOffset;
ILuint AuthInfoOffset;
char HeadString[HEAD_STRING_LEN];
OILHEAD;

ByteHead: This is a string that spells "OIL" (with the terminating zero).

MagicNum: This unsigned long "magic" number is 0x693D71 (or 6897009 in decimal format). Do not ask how this number was generated, as it was a horrid process that noone should ever submit themselves to.

Version: Simply states what version of the .oil format this file is. Unless the .oil format undergoes some major revision, more than likely, this number will stay at 1.

Numl mages: Since the .oil format supports animation, this value is the number of images in the entire file, minus mipmaps, as they are considered "subsets" of an image. DirOffset: Offset from the beginning of the file to the directory. The directory will be explained later in this document.

AuthInfoOffset: Currently means nothing, but set it to 0 always, as it will point to the author information in the future.

HeadString: This is a human-readable string that just describes the type of file it is. If you want to make absolutely certain it's an .oil file and aren't convinced up to now, check this string. You can skip it if you want -- just skip to DirOffset. The string is currently:

"This is a graphics file based on the Open Image Library file format specification." The length of this string is 83 bytes long (HEAD_STRING_LEN) and includes the terminating null character.

The Directory

To accomodate for animation quite easily, .oil files have a directory at DirOffset of the OILHEAD struct. This directory basically just tells where all of the images are located throughout the file. With this kind of system, there is no need to keep images in order in the file, though it is probably desirable for sequential access. You can even put the directory at the end of the file if you so desire. There are as many directory entries as there are number of images, so use the NumImages member of the OILHEAD struct to determine how many directory entries to load. The directory entry is described as such:

typedef struct DIRENT

```
{
      char Name[DIRNAME_LEN];
      ILuint Offset;
      ILuint ImageSize;
```

} DIRENT;

DIRNAME_LEN is 255 characters, and Name is the filename of the file that this image was taken from or even just the regular name of this image. There is no significance to this name, except as a convenience to the author.

Offset is the number of bytes from the beginning of the file to this image. ImageSize is the total size in bytes of the image, including mipmaps and anything else that may be present in the image.

The Image

Finally, we are down to the image itself. An image begins with its own little header:

typedef struct IMAGEHEAD

{

ILuint Width; ILuint Height: ILuint Depth; ILubyte NumChan; ILubyte Bpc; ILubyte Type; ILubyte Compression; ILubyte NumMipmaps; ILuint Duration; ILuint SizeOfData;

} IMAGEHEAD;

Width: Specifies the number of pixels in the x direction.

Height: Specifies the number of pixels in the y direction.

Depth: Specifies the number of pixels in the z direction.

NumChan: Number of colour channels per pixel -- typically equated to bytes per pixel

(or bits per pixel / 8). This number is usually 1, 3 or 4, but any number is theoretically support in the format, though support for it will not be available in any immediate fashion.

Bpc: Bytes per channel -- usually, this is 1, showing that each channel only occupies one byte (one byte for red, one for green, one for blue, etc.). The other common value for this field is 2, usually signifying 64 bits per pixel.

Type: Type is what type the image format is.

- If Type is 1, then the image has a palette.
- If Type is 2, then the image is only luminance values (greyscale).
- If Type is 3, then the image's data is in bgr (blue-green-red) format.
- If Type is 4, then the image's data is in bgra (blue-green-red-alpha) format.

Compression: Tells how the image data has been compressed. This field is what allows us to have virtually any kind of compression. Applications can even try to compress an image various ways before deciding on the best compression style for that particular image before compressing the image. With this field in place, we even have the option of lossy compression! The .oil specifications were designed with lossless compression in mind, but lossy compression may be ideally suited to certain types of images. There are currently three "official" compression schemes right now:

Compression Type:

- 0: No compression. Image data is to be read directly.
- 1: Run-length encoding. This version of rle is adapted from the .tga specification, which can be found at Wotsit's Format.

3: zlib compression. Just uses the uncompress and compress functions from zlib. zlib can be found at the zlib Homepage.

Source examples for all three of these can be found in the OpenIL sources, in oil.c.

NumMipmaps: Tells how many mipmaps immediately follow the image data. These are discussed in greater detail later in this document.

Duration: Specifies the number of milliseconds this image ("frame") should be displayed if part of an animation.

SizeOfData: Actual size of the image data on disk. This is the compressed size, if the image was compressed or, if not, is the size of the image data in memory and on disk. This field is particularly useful for skipping the correct number of bytes if you do not understand the compression type used in this image (such as new compression engines being used in future versions of OpenIL or other programs). The main use of this field though is for decompression of the image data, because you don't want to read to much when decompressing, so you don't overstep an array's boundaries.

Palettes

Only if the Type field of the image's header (IMAGEHEAD) is 1, then the image has a palette. The palette is always in bgra (blue-green-red-alpha) format. Immediately following the SizeOfData member of the image's header is the size of the entire palette in number of bytes as an unsigned long. For instance, if there are 256 palette entries, at 4 bytes per entry (bgra), 1024 should be written here. If the Type field of the image's header is not 1, this unsigned long value is not present.

Image Data

All multichannel image data is in blue-green-red format instead of red-green-blue, like some other image formats. The data is interleaved, meaning that we do not separate data into channels. In other words, our data looks like bgrbgrbgr instead of bbbgggrrr. Luminance data (type 2) is just read as a series of values, as is colour indexed data. How many bytes you read per pixel is dependent on both the number of channels and the bytes per channel. Just multiply these two values to determine how many bytes you must read per pixel. For programs that can only make sense out of one byte per channel, assume that the data is only in the top byte.

Mipmaps

Immediately following the (compressed or uncompressed) image data is the mipmaps. Mipmaps have the exact same format as their parents and even share the same image header, though the Duration and NumMipmaps members are ignored for mipmaps. The duration of the mipmap is the same duration as its parent, and mipmaps are not allowed to have mipmaps of their own.

That should be all for the .oil format. Any comments, questions or suggestions should be sent to Denton Woods.



library.

About

News

Download

SourceForge Home

Documentation

Tutorials

Logos

Links

Projects

Contact Us

A full featured cross-platform image

Beginner's Step-by-Step Tutorial Last Revised: 11:13 PM 12/20/2000

The task of using OpenIL may seem daunting at first, with the multitudes of functions available, but OpenIL is actually relatively easy to use. This tutorial will show you how to create a simple OpenIL application to load, save and display a variety of images.

Checking Versions

This is a critical first step for any well-written application that utilizes OpenIL. With almost all compilers supported, OpenIL is generated as a shared library. Even though the function names may all be the same, earlier versions of OpenIL may have inconsistencies that render your application unuseable. Bugfixes are constantly introduced to try to make OpenIL the best image library ever. The drawback to shared libraries is that a user may inadvertently (or purposefully) replace a newer version of OpenIL with an older version than your application was designed for. Luckily, OpenIL provides version mechanisms to check versions -- ilGetInteger, iluGetInteger and ilutGetInteger. There are #defines in all three libraries that provide the version number your application was compiled with to check against the version number returned by their respective GetInteger functions: IL_VERSION, ILU_VERSION and ILUT_VERSION.

Example of version checking. if (ilGetInteger(IL_VERSION_NUM) < IL_VERSION || iluGetInteger(ILU_VERSION_NUM) < ILU_VERSION)</pre> ilutGetInteger(ILUT_VERSION_NUM) < ILUT_VERSION) {</pre> printf("OpenIL version is different...exiting!\n"); return 1; }

Initializing the Library

With compilers that support shared libraries, no initialization is required, as OpenIL is automatically intialized when it is loaded by an application. Initialization is recommended, though, to ease any porting troubles. Plus, it is only one additional line of source. All that is needed to initialize OpenIL is to call ilStartup. No parameters are even needed.

I mage Names

OpenIL's image name system is virtually identical to OpenGL's texture name system. First, you need an image name variable:

ILuint ImageName;

Next, generate an image name to be put in this variable:

ilGenImages(1, &ImageName);

Now bind this image name so that OpenIL performs all subsequent operations on this

image:

ilBindImage(ImageName);

Creating images is as simple as that. No messy pointers or anything else to mess with. To get an in-depth explanation of image names, read the tutorial on them.

Loading an Image

Loading an image is as simple as it can be with OpenIL. For most programs, a simple call to ilLoadImage will suffice. IF the image was not loaded due to any of various reasons, ilLoadImage returns false, else it returns true if the image was successfully loaded.

Code for loading an image. ilLoadImage("monkey.tga");

Saving an Image

Saving an image is just as easy as loading an image via OpenIL. Just call ilSaveImage with the desired filename as the only parameter. If OpenIL could not save the image, ilSaveImage returns false, else it returns true. By default, OpenIL will refuse to overwrite any images that already exist on the harddrive to prevent from overwriting important data. To change this behaviour to allow overwriting of files, use ilEnable with the IL_FILE_OVERWRITE parameter.

Code for saving an image. ilEnable(IL_FILE_OVERWRITE); ilSaveImage("llama.jpg");

Checking for Errors

Occassionally, errors may occur in OpenIL, such as an image not being loaded. If an OpenIL function returns indicating an error (e.g. returns false from a function that returns an ILboolean), an error code is set internally in OpenIL and may be retrieved via ilGetError. Usually, the code is quite specific about what kind of error occurred. OpenIL maintains an error stack (usually 32 errors deep) so that if more than one error is set, an error doesn't get "lost". When you call ilGetError, the last error set is popped off of the stack. If no error has occurred, or all the errors have been popped off of the stack, ilGetError returns IL_NO_ERROR.

For a more in-depth discussion of errors, read the tutorial on them.

Code for detecting an error. ILenum Error; Error = ilGetError();

Image Information

Of course OpenIL would be pretty useless if you could not retrieve information about the image somehow. ilGetInteger serves this purpose very well, allowing you to know pretty much everything about an image. Some useful values to pass as parameters to ilGetInteger are IL_IMAGE_WIDTH, IL_IMAGE_HEIGHT and IL_IMAGE_BPP. All of these and more are defined in OpenIL's il.h header and in the ilGetInteger documentation.

Code for getting an image's width and height. ILuint Width, Height; Width = ilGetInteger(IL_IMAGE_WIDTH); Height = ilGetInteger(IL_IMAGE_HEIGHT);

Image Data

To get a pointer to the image data for your own use, make a call to ilGetData. ilGetData returns a direct pointer to the image data. Do not try to free this pointer when you are done with it, as it does not point to a copy of the image data but to the actual image data. The image data is freed when you delete the image.

Code for getting the image data. ILubyte *Data = ilGetData();

Display the Image

OpenIL supports several different APIs for displaying an image through ilut. Right now, we will only focus on OpenGL though, as OpenIL is OpenGL's bastard sibling. Since ilut is a separate API, you could manually send data to OpenIL just as ilut does, though it would require more code and time from you (similar to writing your own image routines =).

For a much more in-depth discussion of using OpenIL with OpenGL, read this tutorial.

Before you call any ilut functions dealing with OpenGL and after you have initialized OpenGL, you *must* call ilutRenderer with the ILUT_OPENGL parameter to initialize ilut correctly.

Most applications will then only need to call ilutGLBindTexImage to get a corresponding OpenGL texture from the OpenIL image. If you only need to use the OpenGL texture and not the OpenIL image after this, it is safe to delete the image.

Example of getting an OpenGL texture. GLuint Texture; Texture = ilutGLBindTexImage().

Deleting an Image

To delete an image when you are through with it, call ilDeleteImages. ilDeleteImages frees all data and makes that image name available for use in the future. ilDeleteImages has a syntax exactly like ilGenImages, and the functions are each other's complement.

Example of deleting an image. iIDeleteImages(1, &ImageName);

WELCOME

There are a lot of good and bad documents on how to code, this is another. This is specifically targeted at the game development community, which suffers from two serious evils in coding : the belief that you're in such a hurry to meet deadlines that you don't have time for clean coding practices, and the belief that every bit of code needs to be so optimized that you can't afford clean coding practices. Both of these are wrong, and I'll try to convince you of that. In fact, these two mistakes make game developers some of the worst coders in the industry. There are hardly any programming disciplines (databases, operating systems, applications, web development, etc.) where people swear by the false and old tenent that C++ is too costly, so they'll stick to C, but I've heard it many times from gamers.

There's an evil irony in that the people who would take the time to read a document like this are probably already the "good" ones, because it means they're actively trying to improve their work. The "bad" ones will think themselves above this sort of thing, or just not be interested. That's a big mistake. Programming is all about efficiency, so any new trick or technique you can find to improve your efficiency or that of the team is a huge bonus. You should be reading books on programming, like Scott Meyer's "Effective C++", they really do have new and valuable things to say.

As a manager of programmers, or just a programmer on a team, you must think : what is the job of a programmer? It's not just to write code that "gets the job done" - it's to write code which will result in the entire project being finished well and on time. That means that programmers need to write code which is efficient, easy to debug, easy to modify and extend, and easy for other coders to understand. These later parts are just as important as efficiency, and are often overlooked or not enforced.

EFFICIENCY IN THE RIGHT WAY

When I arrived at Eclipse, I was fond of using plain C and doing nasty things like

if (*(++ptr) == (i = j)) { ...

because I had found in my early days that these kind of obfuscated expressions would turn into more efficient compiled code (I also took a perverse pleasure in it). Dave Stafford wisely told me to stop. He pointed out a key tenet of optimization which I was aware of but had't fully assimilated : if the code is really that important to speed, then you should write it in assembly language or something similar; if it's not, then it should be written for clarity, not efficiency.

This is a form of the old Knuth 80-20 rule; 20% of the code takes 80% of the execution time (hence, optimize it!), while 80% of the code should just be written for clarity and ease of maintenance. Part of the problem in game programming is that most of us started out on Apple 2's or Amigas, or 286 PC's, where the CPU was so slow that you really did have to worry about optimizing all parts of the code. Hence we've got bad habits. Another problem is that C and C++ optimizers used to be very bad. They're not anymore. In fact, I challenge you to write assembly using plain integer instructions which is faster than optimized C code. It's possible, but it's very hard (optimizers still aren't so great with floating point, and of course if you use multimedia extensions you can win).

Modern optimizers are so great, that code written for clarity can often end up faster than code written for speed. That's because it gives the optimizer a better chance to figure out what's going on and do the right thing. I'll go through a lot of specific cases of this later; in all cases, I'll be using the C syntax to make the operation of the code more blatant and restrictive and precise, and it will result in better optimization. Now there's the point of algorithmic efficiency versus tight code. The former is *MANY* orders of magnitude more important. If you have a brute force string matcher written in assembly, I can beat it using a clean C++ implementation of a suffix-trie searcher. Of course, algorithms and spot optimization together will always win, but that really takes too much time. Throughout the development process you need to be able to change your algorithms quickly, and too much early optimization can lock you down in a bad technique. I've spent a lot of wasted time optimizing, because no matter how tight you make a loop to draw a font onto the backbuffer, it'll still be too slow and you'll have to just render the font using sprites and textures - a much better algorithm. Even as recent as a few months ago, we spent a bunch of time here at Surreal spot-optimizing our landscape texture generator, only to find it was still too slow, so we threw it out and came up with a new algorithm.

Related to this is the matter of C++ versus C. Many people are afraid of C++, and they shouldn't be. Instead, they should simply learn about it, what's going on, and how to make sure that it's efficient. Using C++ can greatly improve the clarity, cleanliness, maintainability, abstraction and modularization of code. That's not to say that you can't do all those things with C, it's just much easier with C++. So, the advantages are obvious.

How C++ can cause inefficiency :

1. virtual functions. Yes, they are a slight overhead. However, you should probably not be calling virtual functions in any of your tight loops. For example, your Vector class should not use virtual functions. Virtual functions are useful in class heirarchies where you're abstracting the child relationship, and this should only be done on your top level classes, which are used in the 80% of the code that don't need optimization. Hence, used correctly, virtual functions are no problem.

2. exception handling. This is another overhead, but it's quite easy to get rid of : just disable it. You probably weren't using it anyway.

3. implicit class construction. This is a nasty one, and I'll talk about it more. Basically, this happens when a class conversion is performed by C++, or when you return a class by value from a function. This can be a nasty little inefficiency, but if you do things right, you can avoid it. I'll talk more about it in my list of specific tips, but there are a few basic keys : A. use the "explicit" keyword on constructors, and B. don't define any functions that return a class by value (this includes things like "operator +").

There are some more evils to C++, generally caused by people who are enamored of the features but not aware of the costs, or forgetting what the real point is : clarity and encapsulation. These evils are things like:

1. over-use of pass-by-address. This can cause clarity problems because it's not obvious to the caller that the value he's passing in is being changed. It can also lead (or is caused by) that pass-by-address is somehow "safer" than pointers. It's not, for example addresses can point to null (rarely) and can point into invalid space (e.g., if the object they pointed to was freed and the address remained).

2. over-use of proxy types and templates, derived classes and operator overloads. In general, all of these things should only be used where really necessary and/or natural. For example, an operator ++ that draws a polygon is not wise. You should accomplish your goal with the simplest possible machinery.

Good class design can actually provide the biggest improvement to efficiency possible these days : better memory access patterns. On all the modern game development platforms, cache missses are really the most expensive thing you can do (CPU's are very fast at math these days). Good encapsulation of classes lets you replace the data members with memory-use optimized forms that may be quite nasty (such as run-time compressed data) but all opaqued and hidden away in the class implementation. Thus a client may have no idea that the integer he just requested was actually stored in only two bits.

The final rule of efficiency is to test it, and to examine the assembler. The latter is something that people don't do enough. Say you write some C++ and you're pretty sure that your operators and proxy classes are getting optimized out - well, don't be "pretty sure", tell the compiler to output the assembly and have a look, see what's really happening. You should never write obfuscated code for efficiency purposes unless you have hard proof that it makes a big difference.

GOOD CODING PRACTICE, WHY IT'S WORTH IT

When you start working on someone else's code, perhaps fixing a bug or adding a feature, I'm sure you wish that it was well commented, with clear variable names, and small function bodies.

Bad code results in near-constant debugging, due to programmers' inability to understand how functions are supposed to be used, or unexpected side-effects of changing some un-protected variables. Not only does this slow down development, it makes programmers miserable, and miserable programmers don't write good code.

One of the worst things about bad code is that it spreads. You might hope that it could be contained, and new coders could write better modules into the engine, but this rarely happens. Instead, all new code which refers to the bad old modules inherets the accesses to public variables and unclear function names and duplicated code. Furthermore, good coders working on bad code get frustrated and don't want to spend any excess time in that portion of the code base. The result is that they do shoddy rushed jobs; they also are usually loathe to fix bugs or add enhancements to the bad portions.

1. COMMENTING & CLARITY

Commenting is so obvious and important, there's no reason not to do it. It may take a little more time as you're working, but it'll end up saving hours if not days in debugging and frustration in the future. Comments are especially important when there is some strange "gotcha" or side-effects which are not obvious.

Header files should be commented, with descriptions of each function describing its operation, and especially noting side-effects or inefficiencies. You should consider using an automatic document generator (like Doxygen, etc.) in which case you'll want to comment each function using a style compatible with your documentation tool.

When you implement something and aren't sure about it, or know it's not quite right, you should mark it with a special comment. Also, if you do something lazily or inefficiently, you should comment that and also indicate so in the header. You should use special searchable tags for these, like "@@" and "^~^" so that you can easily find them later.

The key here is that you should think of every function you write as a "service" which is provided to the coders (even if that coder is you). When you later want to use that service, you need to know how to use it and what it does, and how it will effect your code (eg. is it very fast? is it very slow? can it fail in a bad way? can it require user input?).

Another important part of commenting is writing your code in a way that comments itself. If you have some strange self-consistency requirements, add some assert()'s; they not only are useful for debugging, but provide documentation of the interrelationship of your variables. For example, if you have variables like 'counter' and 'counterModulo7' then you should sprinkle in 'assert(counterModulo7 == (counter%7));' Another way to self-document code is through use of descriptive function names and variables. For example, small variable names like 'i' should only be used for loop counters that don't have any other meaning inside the loop.

More ways of self-commenting code include the use of the 'const' directive which lets users know if a field will be modified (and also helps optimization), and by making complicated tasks have complicated names. For example, even though it's "natural" for a matrix to have an "operator *=" multiplication method, I choose not to implement it and make clients call "Multiply()", because I want to make it absolutely clear to users what they're doing when they perform something that expensive.

A little more on the 'const' directive : in some compilers, const can improve optimization alot, because it lets the compiler know that the variable can be stored in any way. For example, if you have a variable which is on the heap (not the stack), the compiler cannot cache it out in a register unless it is const, because it must assume that any time you write to a pointer you might be writing to the memory where that variable is stored. Not 'const'-ing is also definitely contagious, because you cannot call a function which is improperly consted from a function which is consted (without casting). As a side note, when a class member function doesn't modify the "essence" of a class, then it should be declared as if the class were const, with internal casts when necessary. Also, whenever you cast to non-const to modify a value, use the const_cast<> to make it clear why you're casting. You may think this casting is uglier and requires more typing, and you're right, but it should - consting is evil and it should be very apparent to the eye and the fingers when it happens.

Make functions minimal, and make them do only what they say. If you have a long function, it should probably be broken up into smaller functions which each have a specific task. This helps debugging (because you can test each function independently) and re-use, because those little functions may be useful elsewhere, as well as helping readability. Along with this goes the fact that the function can then be easily described by their name. The opposite of this is large functions that do lots of things as well as doing things that are not obvious, such as changing global or static variables. These result in bugs that are hard to track down.

2. MODULARIZATION

Modularization is key to efficient development. It allows one programmer to work on a module of the code base without breaking or involving other sections. Basically, it lets your coding team work like a multi-processor machine; when the code base is not modular (eg. tangled up with dependencies) your coders synchronize, eg. can't work independently.

Modularization is a large topic and more difficult than you may think. It hinges on good class design in C++. Classes should be a minimal implementation of their natural function. If a class is quite complex, perhaps it should be separated into a more fundamental base class and a derived class; put these in different headers so that other modules only need to include what they actually use. The class "interface", that is the public functions it provides, should not lock down it's implementation. For example, accessors that return the member variables are only slightly better than providing access to those variables directly. Which of course brings me to a point I perhaps glossed over : of course those member variables must be private, because making them public lets anyone use them, which locks down the implementation of the class indefinitely.

Leaving classes the freedom to change is very very important. It lets you change your mind about the implementation later if you need to, which is almost always the case. For example, if you had an old 3d engine with a Mesh class which held lots of individual triangles with properties, you might now want to replace it with a Mesh that held a triangle strip - you cannot do that if the old variables are public, because your whole engine may be tangled up in accesses to that class. With a good class interface, you should be able to change the implementation without touching any of the code that depends on that class. In an ideal construction, that includes classes that derive from the one you change, but that may be impossible. Avoid Get() accessors, or at least discourage their use.

Another part of modularization is simply splitting things into separate files and headers. This improves compile times (which is very important) by letting files only include the interfaces they really need. Note that hiding the implementation of classes also improves compile times. For example, any 3d engine should hide the API of the graphics architecture it's running on, so that only a few files in the engine actually need to parse "d3d.h" or "opengl.h" or whatever. Splitting things up also improves the parrallelism of work by making the source-sharing environment work better (CVS, SourceSafe, etc.).

One nice way to acheive modularization is with helper classes and non-member helper functions. These are *not* friend functions or classes, which should essentially never be used, or used sparingly, since friends break modularization and encapsulation. Non-member helper functions for a class are functions which use only the public interface of a class, and automate common operations. Essentially, any manipulation of a class which happens more than once should go into a helper. The helper functions can be in a separate file and header from the main class. Similarly, any function in the class which could be a helper usually should (the exception is functions which may reasonably some day need to be members if the class was implemented differently). Making helpers non-member functions help to minimize the class interface, which makes the more flexible and basic. It also makes it easier to modify and/or replace, since the core functionality is minimal and the non-member helpers need not be changed. You can use a namespace to wrap the non-member helpers. For example, you might have a class Vector and a namespace VectorHelper. Then you would do things like Vector v; VectorHelper::SetRandom(v); // which would use v.SetMembers(x,y,z);

Helper classes are similar, but useful for larger tasks that have many sub-tasks. The helper class is constructed on an instance of the original class (not deriving, rather taking the original as a parameter) and does operations on it. For example, you might have an Image class. You could construct an ImagePainter class which would act on that Image. It would take functions like airbrush that drew into the image, but modify the Image data only through its public accessors.

AN EXAMPLE

Here's an actual example I just found of the bad code I used to be fond of writing. Let's find all the flaws.

This function counts the number of characters which match in the same location in two strings. The first problem is that I make use of the fact that 'bool' has value 0 or 1 when converted to an integer. That's a no-no : using pecularities of C (especially without commenting it), for no good reason. The next problem is that I didn't const the input pointers correctly. Next, the action of the function is un-documented; someone just seeing the name might not realize that characters must "line up" to be counted as matching. Finally, this method really should be a method of String which compares to another string (eg. it's not modularized; if you like using 'char *' for your strings you could just use a Str:: namespace). Here's a slightly better version :

int String::CountCharsSame(const String & vs) const

Note that we've lost some efficiency; in particular, we've taken code that could be compiled into 'setge' and replaced it with a real branch. First of all, we can't take that last sentence too seriously until we look at the disassembly. Second of all, chances are this function is used rarely so clarity is more important than efficiency.

SPECIFIC TIPS

X. use smart pointers

}

They're great; auto_ptr in the STL is not. The "smart pointer" that I'm enamored with is one that points at a ref-counted object. When the smart pointer is made, a reference is taken, when it's destroyed, the reference is released. All functions that return pointers to that object return smart pointers, which makes you quite thread-safe, since anyone using an object always owns a reference to it (this is the standard "ref before returning" paradigm, which Microsoft's COM uses, for example).

X. no binary operators

Binary operators like "operator +" require construction of a temporary. If you're defining operator +, it should only be on a mathematical class which is used in tight loops (like a Vector or Complex number). Thus, construction of temporaries cannot be tolerated (since most optimizers cannot eliminate constructors, even when they do nothing). Thus, you should only declare left-hand-modifying operators, like "operator +=". On a related note, some people think it's cool and good style to pass through the result of "operator +=" and "operator =". While it is true that passing through makes your operators equivalent to the ones on basic C types, like int, I don't really care to allow coders to do things like "a = b = c". Thus, I generally do not pass through the new value.

X. use deferred declaration of variables; also use additional scoping Late declaration of variables (eg. right before they're used) provides optimization. Similarly, using scoping (that is, adding brackets around the lifetime of variables) provides optimization and helps prevent bugs. For example, a variable's lifetime should generally be explicitly terminated when it becomes invalid (eg. when you delete a pointer, let that pointer go out of scope, and also make sure any references to it go out of scope). Late declaration also improves clarity by letting the user see the type of the variable right near its use.

- X. don't use int's declared in a for() elsewhere It's occasionally nice in C++ to declare the loop iterator right in the loop, like for(int i=0;i<n;i++). This stops being nice if you use 'i' later in the function, outside of the loop (that's now againt ANSI C++, so it's a big no-no, even though most popular compilers still allow it).
- X. initialize members with garbage in null constructors in debug mode. This is a nice little trick for making sure that classes which aren't initialized in their constructor do get initialized before use. For example, in my Vector cl I stuff "invalid float" into the members. That way I can make sure the members initialized before being passed into any other function.
- X. don't overuse inheritance or operator overloads. The classic rule for this is that inheritance means "is a", and if that's not natural then maybe you should use "has a" instead - that is, own an instance of the class. For example at Surreal they had a "Frame" time which derived from "Matrix" and "Vector". Now, this is somewhat natural, because it does occasionally make sense to take the translational component of a Frame and use it as a vector. However, is it really an "is a", or would it work just as well by making Frame have a private member which was a Vector? In this case, the public inheritance caused several

then the operation (Frame * Frame) would actually succeed by converting the seconc Frame into a Vector by inheritance and then applying the transformation.

- X. check for &lhs == &rhs in (operator =) Self-explanatory. Avoid bugs.
- X. check for ¶ms == this in most vector/matrix ops! Do this any time the parameters must be read in order to modify the class (this).
- X. use "explicit" on constructors.

"Explicit" on a constructor means that the constructor cannot be invoked by C++ implicitly (to perform a type conversion). For example, if you have a constructor of class B, B::B(class A), then when you pass a type of class A to a function that takes a class B, you may implicitly create the class B. You essentially neve want this happen, or if you do it'll be just as good to do it explicitly.

X. avoid multiple inheritance

It really confuses C++. Multiple inheritance can almost always be avoided by maki a more fundamental base class, so that your classes always have a tree structure. Occasionally, multiple inheritance is nice. Generally, it's wise to avoid it unle the parent classes have orthogonal functionality (eg. operate in different spaces) It's very bad with common ancestors. It can be a nice thing to do with orthogonal parent classes, for example, you might have Renderables and Collidables, which you can put together via multiple inheritance.

X. inline constructors and initializer lists

This is a little-known optimization trick. First of all, initializer lists are gc because they eliminate doubly setting members. That is, you set a member in the k of the constructor, then that member is first constructed, then set. If you do it the initializer list, that member is constructed with the parameter, and that's it If you can write a constructor which uses only the initializer list, then you can declare it inline. In some cases this will result in the constructor actually bei inlined. Note that most optimizers *still* won't optimize it out even if it's unecessary (see: dont use binary operators) but at least it won't be a function ca

X. virtual destructors

If you have virtual functions in a derived class, then you probably want virtual destructors. A virtual destructor means that if you have a class of type B which cast to a class of type A, then when you delete it you actually get the destructor B. If you think about it, this is really always what you want to happen, and non-destructors of publicly derived classes are just memory leaks waiting to happen.

X. base classes for templates

Templates are like a lot of things in C++ : great if used right, but can be a penalty if used badly or overused. In particular, templates can cause code-bloat increased compiled code size, and increased compile time. One way to reduce this to implement templates in terms of a common base class. For example, if you had & template linked list "t_link", then you might define a linked list base class "lir which is not a template and doesn't contain a data type. You then implement as mulogic as possible in the link class, and make t_link derive from it. Thus, all instances of the t_link class will make use of the shared code from the link class

X. use const !

Did I mention this already? I use const all over, even on temporary variables and function parameters. If you need to modify a function parameter that's passed in by value, make a new variable and read out of the const parameter. This not only produces faster code, it also helps debugging, because the original parameters are left intact for inspection.

X. write function test loops

When you write a difficult new function, you should make a mini program which tests it by throwing all kinds of input at it and confirming the results. This mini program might be left in the main codebase so that you can run it again later and make sure your functions still work. This is part of the larger task of writing modules which are *reliable* so that clients (including yourself) can use them elsewhere and not have to worry about acquiring bugs from the old module.

X. use assert() and do error checking

I can never get enough of assert(). It's awesome for debugging; every function should be blanketed with enough asserts to make sure that the values that come into it are valid, and the values that come out are right. The best types of asserts are self-consistency checks (which verify some implicit relationship between the member variables of a class), and checks of a function's action by performing the same task a different way and comparing results. Along with this goes error checking. Any assert() which is not in a speed-critical location may need to also have a real error case for the release build. All user input and data files should be checked for consistency and handled with proper errors. I've worked places where this was not done, and I spent a lot of time looking for bugs in my code when it was actually an invalid program input that I was passing in and the old code wasn't checking for errors.

X. be careful about using Hungarian notation

(Hungarian notation is the style in which you pre-pend variable names with a description of their type). Hungarian can be nice when used with discrimination. Used excessively, it forms a type of dependence on implementation. That is, if I wish to change the implementation of a class, I may need to change the types of its members, but may not need to rewrite every function that uses those members. For example, if you used "w" for WORD data and "dw" for DWORD data, and you needed to change a flag field from 16 to 32 bits, you would have to change every reference to "wFlags" to "dwFlags". Clearly this goes against the spirit of the notation and the ability to seamlessly change the underlying implementation. (see http://msdn.microsoft.com/library/techart/hunganotat.htm)

Page 8 of 8

Chapter 1 Introduction to OpenGL

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Appreciate in general terms what OpenGL does
- Identify different levels of rendering complexity
- Understand the basic structure of an OpenGL program
- Recognize OpenGL command syntax
- Identify the sequence of operations of the OpenGL rendering pipeline
- Understand in general terms how to animate graphics in an OpenGL program

This chapter introduces OpenGL. It has the following major sections:

- "What Is OpenGL?" explains what OpenGL is, what it does and doesn't do, and how it works.
- "A Smidgen of OpenGL Code" presents a small OpenGL program and briefly discusses it. This section also defines a few basic computer-graphics terms.
- "OpenGL Command Syntax" explains some of the conventions and notations used by OpenGL commands.
- "OpenGL as a State Machine" describes the use of state variables in OpenGL and the commands for querying, enabling, and disabling states.
- "OpenGL Rendering Pipeline" shows a typical sequence of operations for processing geometric and image data.
- "OpenGL-Related Libraries" describes sets of OpenGL-related routines, including an auxiliary library specifically written for this book to simplify programming examples.
- "Animation" explains in general terms how to create pictures on the screen that move.

What Is OpenGL?

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you're using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, you must build up your desired model from a small set of *geometric primitives* - points, lines, and polygons.

A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation. Also, there is a higher-level, object-oriented toolkit, Open Inventor, which is built atop OpenGL, and is available separately for many implementations of OpenGL. (See "OpenGL-Related Libraries" for more information about Open Inventor.)

Now that you know what OpenGL *doesn't* do, here's what it *does* do. Take a look at the color plates - they illustrate typical uses of OpenGL. They show the scene on the cover of this book, *rendered* (which is to say, drawn) by a computer using OpenGL in successively more complicated ways. The following list describes in general terms how these pictures were made.

• "Plate 1" shows the entire scene displayed as a wireframe model - that is, as if all the objects in the scene were made of wire. Each line of wire corresponds to an edge of a primitive (typically a polygon). For example, the surface of the table is constructed from triangular polygons that are positioned like slices of pie.

Note that you can see portions of objects that would be obscured if the objects were solid rather than wireframe. For example, you can see the entire model of the hills outside the window even though most of this model is normally hidden by the wall of the room. The globe appears to be nearly solid because it's composed of hundreds of colored blocks, and you see the wireframe lines for all the edges of all the blocks, even those forming the back side of the globe. The way the globe is constructed gives you an idea of how complex objects can be created by assembling lower-level objects.

- "Plate 2" shows a *depth-cued* version of the same wireframe scene. Note that the lines farther from the eye are dimmer, just as they would be in real life, thereby giving a visual cue of depth. OpenGL uses atmospheric effects (collectively referred to as fog) to achieve depth cueing.
- "Plate 3" shows an *antialiased* version of the wireframe scene. Antialiasing is a technique for reducing the jagged edges (also known as *jaggies*) created when approximating smooth edges using *pixels* short for picture *elements* which are confined to a rectangular grid. Such jaggies

are usually the most visible with near-horizontal or near-vertical lines.

- "Plate 4" shows a *flat-shaded*, *unlit* version of the scene. The objects in the scene are now shown as solid. They appear "flat" in the sense that only one color is used to render each polygon, so they don't appear smoothly rounded. There are no effects from any light sources.
- "Plate 5" shows a *lit, smooth-shaded* version of the scene. Note how the scene looks much more realistic and three-dimensional when the objects are shaded to respond to the light sources in the room as if the objects were smoothly rounded.
- "Plate 6" adds *shadows* and *textures* to the previous version of the scene. Shadows aren't an explicitly defined feature of OpenGL (there is no "shadow command"), but you can create them yourself using the techniques described in Chapter 14. *Texture mapping* allows you to apply a two-dimensional image onto a three-dimensional object. In this scene, the top on the table surface is the most vibrant example of texture mapping. The wood grain on the floor and table surface are all texture mapped, as well as the wallpaper and the toy top (on the table).
- "Plate 7" shows a *motion-blurred* object in the scene. The sphinx (or dog, depending on your Rorschach tendencies) appears to be captured moving forward, leaving a blurred trace of its path of motion.
- "Plate 8" shows the scene as it's drawn for the cover of the book from a different viewpoint. This plate illustrates that the image really is a snapshot of models of three-dimensional objects.
- "Plate 9" brings back the use of fog, which was seen in "Plate 2," to show the presence of smoke particles in the air. Note how the same effect in "Plate 2" now has a more dramatic impact in "Plate 9."
- "Plate 10" shows the *depth-of-field effect*, which simulates the inability of a camera lens to maintain all objects in a photographed scene in focus. The camera focuses on a particular spot in the scene. Objects that are significantly closer or farther than that spot are somewhat blurred.

The color plates give you an idea of the kinds of things you can do with the OpenGL graphics system. The following list briefly describes the major graphics operations which OpenGL performs to render an image on the screen. (See "OpenGL Rendering Pipeline" for detailed information about this order of operations.)

- 1. Construct shapes from geometric primitives, thereby creating mathematical descriptions of objects. (OpenGL considers points, lines, polygons, images, and bitmaps to be primitives.)
- 2. Arrange the objects in three-dimensional space and select the desired vantage point for viewing the composed scene.
- 3. Calculate the color of all the objects. The color might be explicitly assigned by the application, determined from specified lighting conditions, obtained by pasting a texture onto the objects, or some combination of these three actions.
- 4. Convert the mathematical description of objects and their associated color information to pixels on

the screen. This process is called *rasterization*.

During these stages, OpenGL might perform other operations, such as eliminating parts of objects that are hidden by other objects. In addition, after the scene is rasterized but before it's drawn on the screen, you can perform some operations on the pixel data if you want.

In some implementations (such as with the X Window System), OpenGL is designed to work even if the computer that displays the graphics you create isn't the computer that runs your graphics program. This might be the case if you work in a networked computer environment where many computers are connected to one another by a digital network. In this situation, the computer on which your program runs and issues OpenGL drawing commands is called the client, and the computer that receives those commands and performs the drawing is called the server. The format for transmitting OpenGL commands (called the *protocol*) from the client to the server is always the same, so OpenGL programs can work across a network even if the client and server are different kinds of computers. If an OpenGL program isn't running across a network, then there's only one computer, and it is both the client and the server.

A Smidgen of OpenGL Code

Because you can do so many things with the OpenGL graphics system, an OpenGL program can be complicated. However, the basic structure of a useful program can be simple: Its tasks are to initialize certain states that control how OpenGL renders and to specify objects to be rendered.

Before you look at some OpenGL code, let's go over a few terms. *Rendering*, which you've already seen used, is the process by which a computer creates images from models. These *models*, or objects, are constructed from geometric primitives - points, lines, and polygons - that are specified by their vertices.

The final rendered image consists of pixels drawn on the screen; a pixel is the smallest visible element the display hardware can put on the screen. Information about the pixels (for instance, what color they're supposed to be) is organized in memory into bitplanes. A bitplane is an area of memory that holds one bit of information for every pixel on the screen; the bit might indicate how red a particular pixel is supposed to be, for example. The bitplanes are themselves organized into a *framebuffer*, which holds all the information that the graphics display needs to control the color and intensity of all the pixels on the screen.

Now look at what an OpenGL program might look like. Example 1-1 renders a white rectangle on a black background, as shown in Figure 1-1.

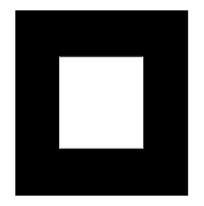


Figure 1-1 : White Rectangle on a Black Background

Example 1-1 : Chunk of OpenGL Code

```
# include <whateverYouNeed.h>
main() {
    InitializeAWindowPlease();
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
      glFlush();
    UpdateTheWindowAndCheckForEvents();
```

```
}
```

The first line of the **main**() routine initializes a *window* on the screen: The **InitializeAWindowPlease**() routine is meant as a placeholder for window system-specific routines, which are generally not OpenGL calls. The next two lines are OpenGL commands that clear the window to black: **glClearColor**() establishes what color the window will be cleared to, and **glClear**() actually clears the window. Once the clearing color is set, the window is cleared to that color whenever **glClear**() is called. This clearing color can be changed with another call to **glClearColor**(). Similarly, the **glColor3f**() command establishes what color to use for drawing objects - in this case, the color is white. All objects drawn after this point use this color, until it's changed with another call to set the color.

The next OpenGL command used in the program, **glOrtho**(), specifies the coordinate system OpenGL assumes as it draws the final image and how the image gets mapped to the screen. The next calls, which are bracketed by **glBegin**() and **glEnd**(), define the object to be drawn - in this example, a polygon with four vertices. The polygon's "corners" are defined by the **glVertex3f**() commands. As you might be able to guess from the arguments, which are (*x*, *y*, *z*) coordinates, the polygon is a rectangle on the z=0 plane.

Finally, **glFlush()** ensures that the drawing commands are actually executed rather than stored in a *buffer* awaiting additional OpenGL commands. The **UpdateTheWindowAndCheckForEvents()** placeholder routine manages the contents of the window and begins event processing.

Actually, this piece of OpenGL code isn't well structured. You may be asking, "What happens if I try to move or resize the window?" Or, "Do I need to reset the coordinate system each time I draw the rectangle?" Later in this chapter, you will see replacements for both **InitializeAWindowPlease()** and **UpdateTheWindowAndCheckForEvents()** that actually work but will require restructuring the code to make it efficient.

OpenGL Command Syntax

As you might have observed from the simple program in the previous section, OpenGL commands use the prefix **gl** and initial capital letters for each word making up the command name (recall **glClearColor**(), for example). Similarly, OpenGL defined constants begin with GL_, use all capital letters, and use underscores to separate words (like GL_COLOR_BUFFER_BIT).

You might also have noticed some seemingly extraneous letters appended to some command names (for example, the **3f** in **glColor3f**() **and glVertex3f**()). It's true that the **Color** part of the command name **glColor3f**() is enough to define the command as one that sets the current color. However, more than one such command has been defined so that you can use different types of arguments. In particular, the **3** part of the suffix indicates that three arguments are given; another version of the **Color** command takes four arguments. The **f** part of the suffix indicates that the arguments are floating-point numbers. Having different formats allows OpenGL to accept the user's data in his or her own data format.

Some OpenGL commands accept as many as 8 different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are shown in Table 1-1, along with the corresponding OpenGL type definitions. The particular implementation of OpenGL that you're using might not follow this scheme exactly; an implementation in C++ or Ada, for example, wouldn't need to.

Table 1-1 : Command Suffixes and Argument Data Types

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Thus, the two commands

```
glVertex2i(1, 3);
glVertex2f(1.0, 3.0);
```

are equivalent, except that the first specifies the vertex's coordinates as 32-bit integers, and the second specifies them as single-precision floating-point numbers.

Note: Implementations of OpenGL have leeway in selecting which C data type to use to represent OpenGL data types. If you resolutely use the OpenGL defined data types throughout your application, you will avoid mismatched types when porting your code between different implementations.

Some OpenGL commands can take a final letter \mathbf{v} , which indicates that the command takes a pointer to a vector (or array) of values rather than a series of individual arguments. Many commands have both vector and nonvector versions, but some commands accept only individual arguments and others require that at least some of the arguments be specified as a vector. The following lines show how you might use a vector and a nonvector version of the command that sets the current color:

```
glColor3f(1.0, 0.0, 0.0);
GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

Finally, OpenGL defines the typedef GLvoid. This is most often used for OpenGL commands that accept pointers to arrays of values.

In the rest of this guide (except in actual code examples), OpenGL commands are referred to by their base names only, and an asterisk is included to indicate that there may be more to the command name. For example, **glColor*()** stands for all variations of the command you use to set the current color. If we want to make a specific point about one version of a particular command, we include the suffix necessary to define that version. For example, **glVertex*v()** refers to all the vector versions of the command you use to specify vertices.

OpenGL as a State Machine

OpenGL is a state machine. You put it into various states (or modes) that then remain in effect until you change them. As you've already seen, the current color is a state variable. You can set the current color to white, red, or any other color, and thereafter every object is drawn with that color until you set the current color to something else. The current color is only one of many state variables that OpenGL maintains. Others control such things as the current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes, pixel-packing conventions, positions and characteristics of lights, and material properties of the objects being drawn. Many state variables refer to modes that are enabled or disabled with the command **glEnable(**) or **glDisable(**).

Each state variable or mode has a default value, and at any point you can query the system for each variable's current value. Typically, you use one of the six following commands to do this: glGetBooleanv(), glGetDoublev(), glGetFloatv(), glGetIntegerv(), glGetPointerv(), or glIsEnabled(). Which of these commands you select depends on what data type you want the answer to be given in. Some state variables have a more specific query command (such as glGetLight*(), glGetError(), or glGetPolygonStipple()). In addition, you can save a collection of state variables on an attribute stack with glPushAttrib() or glPushClientAttrib(), temporarily modify them, and later restore the values with glPopAttrib() or glPopClientAttrib(). For temporary state changes, you should use these commands rather than any of the query commands, since they're likely to be more efficient.

See Appendix B for the complete list of state variables you can query. For each variable, the appendix also lists a suggested **glGet***() command that returns the variable's value, the attribute class to which it belongs, and the variable's default value.

OpenGL Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering, as shown in Figure 1-2, is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do.

If you are new to three-dimensional graphics, the upcoming description may seem like drinking water out of a fire hose. You can skim this now, but come back to Figure 1-2 as you go through each chapter in this book.

The following diagram shows the Henry Ford assembly line approach, which OpenGL takes to processing data. Geometric data (vertices, lines, and polygons) follow the path through the row of boxes

that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer.

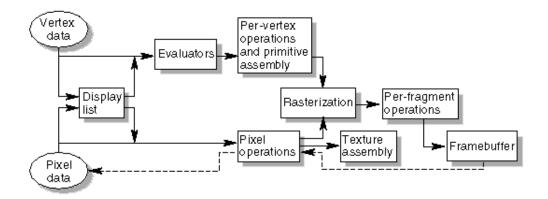


Figure 1-2 : Order of Operations

Now you'll see more detail about the key stages in the OpenGL rendering pipeline.

Display Lists

All data, whether it describes geometry or pixels, can be saved in a *display list* for current or later use. (The alternative to retaining data in a display list is processing the data immediately - also known as *immediate mode*.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode. (See Chapter 7 for more information about display lists.)

Evaluators

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points. (See Chapter 12 to learn more about evaluators.)

Per-Vertex Operations

For vertex data, next is the "per-vertex operations" stage, which converts the vertices into primitives. Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. (See Chapter 3 for details about the transformation matrices.)

If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

Primitive Assembly

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped.

In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines. (See "Polygon Details" in Chapter 2.)

The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

Pixel Operations

While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. (See "Imaging Pipeline" in Chapter 8.)

If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory.

There are special pixel copy operations to copy data in the framebuffer to other parts of the framebuffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the framebuffer.

Texture Assembly

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them.

Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource. (See Chapter 9.)

Rasterization

Rasterization is the conversion of both geometric and pixel data into *fragments*. Each fragment square corresponds to a pixel in the framebuffer. Line and polygon stipples, line width, point size, shading

model, and coverage calculations to support antialiasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

Fragment Operations

Before values are actually stored into the framebuffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled.

The first operation which may be encountered is texturing, where a texel (texture element) is generated from texture memory for each fragment and applied to the fragment. Then fog calculations may be applied, followed by the scissor test, the alpha test, the stencil test, and the depth-buffer test (the depth buffer is for hidden-surface removal). Failing an enabled test may end the continued processing of a fragment's square. Then, blending, dithering, logical operation, and masking by a bitmask may be performed. (See Chapter 6 and Chapter 10) Finally, the thoroughly processedfragment is drawn into the appropriate buffer, where it has finally advanced to be a pixel and achieved its final resting place.

OpenGL-Related Libraries

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow you to simplify your programming tasks, including the following:

- The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. Portions of the GLU are described in the *OpenGL Reference Manual*. The more useful GLU routines are described in this guide, where they're relevant to the topic being discussed, such as in all of Chapter 11 and in the section "The GLU NURBS Interface" in Chapter 12. GLU routines use the prefix **glu**.
- For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix **glX**. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. All WGL routines use the prefix **wgl**. For IBM OS/2, the PGL is the Presentation Manager to OpenGL interface, and its routines use the prefix **pgl**.

All these window system extension libraries are described in more detail in both Appendix C. In addition, the GLX routines are also described in the *OpenGL Reference Manual*.

• The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs. GLUT is the subject of the next section, and it's described in more detail in Mark Kilgard's book *OpenGL Programming for the X Window System* (ISBN 0-201-48359-9). GLUT routines use the prefix **glut.** "How to Obtain

the Sample Code" in the Preface describes how to obtain the source code for GLUT, using ftp.

• Open Inventor is an object-oriented toolkit based on OpenGL which provides objects and methods for creating interactive three-dimensional graphics applications. Open Inventor, which is written in C++, provides prebuilt objects and a built-in event model for user interaction, high-level application components for creating and editing three-dimensional scenes, and the ability to print objects and exchange data in other graphics formats. Open Inventor is separate from OpenGL.

Include Files

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which requires inclusion of the glu.h header file. So almost every OpenGL source file begins with

#include <GL/gl.h>
#include <GL/glu.h>

If you are directly accessing a window interface library to support OpenGL, such as GLX, AGL, PGL, or WGL, you must include additional header files. For example, if you are calling GLX, you may need to add these lines to your code

#include <X11/Xlib.h>
#include <GL/glx.h>

If you are using GLUT for managing your window manager tasks, you should include

#include <GL/glut.h>

Note that glut.h includes gl.h, glu.h, and glx.h automatically, so including all three files is redundant. GLUT for Microsoft Windows includes the appropriate header file to access WGL.

GLUT, the OpenGL Utility Toolkit

As you know, OpenGL contains rendering commands but is designed to be independent of any window system or operating system. Consequently, it contains no commands for opening windows or reading events from the keyboard or mouse. Unfortunately, it's impossible to write a complete graphics program without at least opening a window, and most interesting programs require a bit of user input or other services from the operating system or window system. In many cases, complete programs make the most interesting examples, so this book uses GLUT to simplify opening windows, detecting input, and so on. If you have an implementation of OpenGL and GLUT on your system, the examples in this book should run without change when linked with them.

In addition, since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), GLUT includes several routines that create more complicated three-dimensional objects such as a sphere, a torus, and a teapot. This way, snapshots of program output can be interesting to look at. (Note that the OpenGL Utility Library, GLU, also has quadrics routines that create some of the same three-dimensional objects as GLUT, such as a sphere, cylinder, or cone.)

GLUT may not be satisfactory for full-featured OpenGL applications, but you may find it a useful

starting point for learning OpenGL. The rest of this section briefly describes a small subset of GLUT routines so that you can follow the programming examples in the rest of this book. (See Appendix D for more details about this subset of GLUT, or see Chapters 4 and 5 of *OpenGL Programming for the X Window System* for information about the rest of GLUT.)

Window Management

Five routines perform tasks necessary to initialize a window.

- **glutInit**(int **argc*, char ***argv*) initializes GLUT and processes any command line arguments (for X, this would be options like -display and -geometry). **glutInit**() should be called before any other GLUT routine.
- **glutInitDisplayMode**(unsigned int *mode*) specifies whether to use an *RGBA* or color-index color model. You can also specify whether you want a single- or double-buffered window. (If you're working in color-index mode, you'll want to load certain colors into the color map; use **glutSetColor**() to do this.) Finally, you can use this routine to indicate that you want the window to have an associated depth, stencil, and/or accumulation buffer. For example, if you want a window with double buffering, the RGBA color model, and a depth buffer, you might call **glutInitDisplayMode**(*GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH*).
- **glutInitWindowPosition**(int *x*, int *y*) specifies the screen location for the upper-left corner of your window.
- **glutInitWindowSize**(int *width*, int *size*) specifies the size, in pixels, of your window.
- int **glutCreateWindow**(char **string*) creates a window with an OpenGL context. It returns a unique identifier for the new window. Be warned: Until **glutMainLoop**() is called (see next section), the window is not yet displayed.

The Display Callback

glutDisplayFunc(void (**func*)(void)) is the first and most important event callback function you will see. Whenever GLUT determines the contents of the window need to be redisplayed, the callback function registered by **glutDisplayFunc**() is executed. Therefore, you should put all the routines you need to redraw the scene in the display callback function.

If your program changes the contents of the window, sometimes you will have to call **glutPostRedisplay**(void), which gives **glutMainLoop**() a nudge to call the registered display callback at its next opportunity.

Running the Program

The very last thing you must do is call **glutMainLoop**(void). All windows that have been created are now shown, and rendering to those windows is now effective. Event processing begins, and the registered display callback is triggered. Once this loop is entered, it is never exited!

Example 1-2 shows how you might use GLUT to create the simple program shown in Example 1-1.

Note the restructuring of the code. To maximize efficiency, operations that need only be called once (setting the background color and coordinate system) are now in a procedure called **init**(). Operations to render (and possibly re-render) the scene are in the **display**() procedure, which is the registered GLUT display callback.

Example 1-2 : Simple OpenGL Program Using GLUT: hello.c

```
#include <GL/ql.h>
#include <GL/glut.h>
void display(void)
{
·
/*
   clear all pixels
                     */
   glClear (GL COLOR BUFFER BIT);
/* draw white polygon (rectangle) with corners at
 * (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
 * /
   glColor3f (1.0, 1.0, 1.0);
    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0);
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
        glVertex3f (0.25, 0.75, 0.0);
   glEnd();
/* don't wait!
   start processing buffered OpenGL routines
 * /
   glFlush ();
}
void init (void)
/*
    select clearing (background) color
                                              */
    glClearColor (0.0, 0.0, 0.0, 0.0);
/* initialize viewing values */
   glMatrixMode(GL PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}
/*
 *
  Declare initial window size, position, and display mode
 *
   (single buffer and RGBA). Open window with "hello"
 *
   in its title bar. Call initialization routines.
 *
   Register callback function to display graphics.
 *
   Enter main loop and process events.
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("hello");
    init ();
    glutDisplayFunc(display);
```

```
glutMainLoop();
return 0; /* ISO C requires main to return int. */
}
```

Handling Input Events

You can use these routines to register callback commands that are invoked when specified events occur.

- **glutReshapeFunc**(void (* *func*)(int w, int h)) indicates what action should be taken when the window is resized.
- **glutKeyboardFunc**(void (**func*)(unsigned char *key*, int *x*, int *y*)) and **glutMouseFunc**(void (**func*)(int *button*, int *state*, int *x*, int *y*)) allow you to link a keyboard key or a mouse button with a routine that's invoked when the key or mouse button is pressed or released.
- **glutMotionFunc**(void (**func*)(int *x*, int *y*)) registers a routine to call back when the mouse is moved while a mouse button is also pressed.

Managing a Background Process

You can specify a function that's to be executed if no other events are pending - for example, when the event loop would otherwise be idle - with **glutIdleFunc**(void (**func*)(void)). This routine takes a pointer to the function as its only argument. Pass in NULL (zero) to disable the execution of the function.

Drawing Three-Dimensional Objects

GLUT includes several routines for drawing these three-dimensional objects:

cone	icosahedron	teapot
cube	octahedron	tetrahedron
dodecahedron	sphere	torus

You can draw these objects as wireframes or as solid shaded objects with surface normals defined. For example, the routines for a cube and a sphere are as follows:

void glutWireCube(GLdouble size);

void glutSolidCube(GLdouble size);

void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);

void **glutSolidSphere**(GLdouble *radius*, GLint *slices*, GLint *stacks*);

All these models are drawn centered at the origin of the world coordinate system. (See for information on the prototypes of all these drawing routines.)

Animation

One of the most exciting things you can do on a graphics computer is draw pictures that move. Whether you're an engineer trying to see all sides of a mechanical part you're designing, a pilot learning to fly an airplane using a simulation, or merely a computer-game aficionado, it's clear that animation is an important part of computer graphics.

In a movie theater, motion is achieved by taking a sequence of pictures and projecting them at 24 per second on the screen. Each frame is moved into position behind the lens, the shutter is opened, and the frame is displayed. The shutter is momentarily closed while the film is advanced to the next frame, then that frame is displayed, and so on. Although you're watching 24 different frames each second, your brain blends them all into a smooth animation. (The old Charlie Chaplin movies were shot at 16 frames per second and are noticeably jerky.) In fact, most modern projectors display each picture twice at a rate of 48 per second to reduce flickering. Computer-graphics screens typically refresh (redraw the picture) approximately 60 to 76 times per second, and some even run at about 120 refreshes per second. Clearly, 60 per second is smoother than 30, and 120 is marginally better than 60. Refresh rates faster than 120, however, are beyond the point of diminishing returns, since the human eye is only so good.

The key reason that motion picture projection works is that each frame is complete when it is displayed. Suppose you try to do computer animation of your million-frame movie with a program like this:

```
open_window();
for (i = 0; i < 1000000; i++) {
    clear_the_window();
    draw_frame(i);
    wait_until_a_24th_of_a_second_is_over();
}
```

If you add the time it takes for your system to clear the screen and to draw a typical frame, this program gives more and more disturbing results depending on how close to 1/24 second it takes to clear and draw. Suppose the drawing takes nearly a full 1/24 second. Items drawn first are visible for the full 1/24 second and present a solid image on the screen; items drawn toward the end are instantly cleared as the program starts on the next frame. They present at best a ghostlike image, since for most of the 1/24 second your eye is viewing the cleared background instead of the items that were unlucky enough to be drawn last. The problem is that this program doesn't display completely drawn frames; instead, you watch the drawing as it happens.

Most OpenGL implementations provide double-buffering - hardware or software that supplies two complete color buffers. One is displayed while the other is being drawn. When the drawing of a frame is complete, the two buffers are swapped, so the one that was being viewed is now used for drawing, and vice versa. This is like a movie projector with only two frames in a loop; while one is being projected on the screen, an artist is desperately erasing and redrawing the frame that's not visible. As long as the artist is quick enough, the viewer notices no difference between this setup and one where all the frames are already drawn and the projector is simply displaying them one after the other. With double-buffering, every frame is shown only when the drawing is complete; the viewer never sees a partially drawn frame.

A modified version of the preceding program that does display smoothly animated graphics might look like this:

```
open_window_in_double_buffer_mode();
for (i = 0; i < 1000000; i++) {
    clear_the_window();
    draw_frame(i);
    swap_the_buffers();
}
```

The Refresh That Pauses

For some OpenGL implementations, in addition to simply swapping the viewable and drawable buffers, the **swap_the_buffers**() routine waits until the current screen refresh period is over so that the previous buffer is completely displayed. This routine also allows the new buffer to be completely displayed, starting from the beginning. Assuming that your system refreshes the display 60 times per second, this means that the fastest frame rate you can achieve is 60 frames per second (*fps*), and if all your frames can be cleared and drawn in under 1/60 second, your animation will run smoothly at that rate.

What often happens on such a system is that the frame is too complicated to draw in 1/60 second, so each frame is displayed more than once. If, for example, it takes 1/45 second to draw a frame, you get 30 fps, and the graphics are idle for 1/30-1/45=1/90 second per frame, or one-third of the time.

In addition, the video refresh rate is constant, which can have some unexpected performance consequences. For example, with the 1/60 second per refresh monitor and a constant frame rate, you can run at 60 fps, 30 fps, 20 fps, 15 fps, 12 fps, and so on (60/1, 60/2, 60/3, 60/4, 60/5, ...). That means that if you're writing an application and gradually adding features (say it's a flight simulator, and you're adding ground scenery), at first each feature you add has no effect on the overall performance - you still get 60 fps. Then, all of a sudden, you add one new feature, and the system can't quite draw the whole thing in 1/60 of a second, so the animation slows from 60 fps to 30 fps because it misses the first possible buffer-swapping time. A similar thing happens when the drawing time per frame is more than 1/30 second - the animation drops from 30 to 20 fps.

If the scene's complexity is close to any of the magic times (1/60 second, 2/60 second, 3/60 second, and so on in this example), then because of random variation, some frames go slightly over the time and some slightly under. Then the frame rate is irregular, which can be visually disturbing. In this case, if you can't simplify the scene so that all the frames are fast enough, it might be better to add an intentional, tiny delay to make sure they all miss, giving a constant, slower, frame rate. If your frames have drastically different complexities, a more sophisticated approach might be necessary.

Motion = Redraw + Swap

The structure of real animation programs does not differ too much from this description. Usually, it is easier to redraw the entire buffer from scratch for each frame than to figure out which parts require redrawing. This is especially true with applications such as three-dimensional flight simulators where a tiny change in the plane's orientation changes the position of everything outside the window.

In most animations, the objects in a scene are simply redrawn with different transformations - the viewpoint of the viewer moves, or a car moves down the road a bit, or an object is rotated slightly. If significant recomputation is required for non-drawing operations, the attainable frame rate often slows down. Keep in mind, however, that the idle time after the **swap_the_buffers()** routine can often be used for such calculations.

OpenGL doesn't have a **swap_the_buffers**() command because the feature might not be available on all hardware and, in any case, it's highly dependent on the window system. For example, if you are using the X Window System and accessing it directly, you might use the following GLX routine:

void glXSwapBuffers(Display *dpy, Window window);

(See Appendix C for equivalent routines for other window systems.)

If you are using the GLUT library, you'll want to call this routine:

void glutSwapBuffers(void);

Example 1-3 illustrates the use of **glutSwapBuffers**() in an example that draws a spinning square as shown in Figure 1-3. The following example also shows how to use GLUT to control an input device and turn on and off an idle function. In this example, the mouse buttons toggle the spinning on and off.

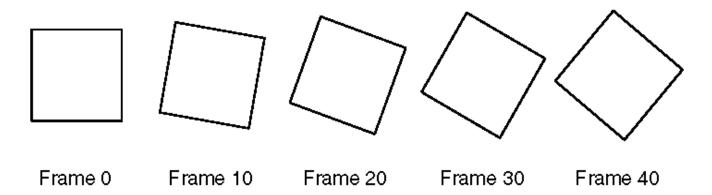


Figure 1-3 : Double-Buffered Rotating Square

Example 1-3 : Double-Buffered Program: double.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <GL/glut.h>
#include <stdlib.h>
static GLfloat spin = 0.0;
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glPushMatrix();
    glRotatef(spin, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
}
```

```
glRectf(-25.0, -25.0, 25.0, 25.0);
   glPopMatrix();
   glutSwapBuffers();
}
void spinDisplay(void)
   spin = spin + 2.0;
   if (spin > 360.0)
      spin = spin - 360.0;
   glutPostRedisplay();
}
void reshape(int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL PROJECTION);
   glLoadIdentity();
   glortho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   glMatrixMode(GL MODELVIEW);
   glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(spinDisplay);
         break;
      case GLUT_MIDDLE_BUTTON:
         if (state == GLUT DOWN)
            glutIdleFunc(NULL);
         break;
      default:
         break;
   }
}
/*
 *
   Request double buffer display mode.
 *
   Register mouse input callback functions
 */
int main(int argc, char** argv)
ł
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

+ +

Publishing Company)

Chapter 2 State Management and Drawing Geometric Objects

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Clear the window to an arbitrary color
- Force any pending drawing to complete
- Draw with any geometric primitive points, lines, and polygons in two or three dimensions
- Turn states on and off and query state variables
- Control the display of those primitives for example, draw dashed lines or outlined polygons
- Specify normal vectors at appropriate points on the surface of solid objects
- Use *vertex arrays* to store and access a lot of geometric data with only a few function calls
- Save and restore several state variables at once

Although you can draw complex and interesting pictures using OpenGL, they're all constructed from a small number of primitive graphical items. This shouldn't be too surprising - look at what Leonardo da Vinci accomplished with just pencils and paintbrushes.

At the highest level of abstraction, there are three basic drawing operations: clearing the window, drawing a geometric object, and drawing a raster object. Raster objects, which include such things as two-dimensional images, bitmaps, and character fonts, are covered in Chapter 8. In this chapter, you learn how to clear the screen and to draw geometric objects, including points, straight lines, and flat polygons.

You might think to yourself, "Wait a minute. I've seen lots of computer graphics in movies and on television, and there are plenty of beautifully shaded curved lines and surfaces. How are those drawn, if all OpenGL can draw are straight lines and flat polygons?" Even the image on the cover of this book includes a round table and objects on the table that have curved surfaces. It turns out that all the curved lines and surfaces you've seen are approximated by large numbers of little flat polygons or straight lines, in much the same way that the globe on the cover is constructed from a large set of rectangular blocks.

+ +

The globe doesn't appear to have a smooth surface because the blocks are relatively large compared to the globe. Later in this chapter, we show you how to construct curved lines and surfaces from lots of small geometric primitives.

This chapter has the following major sections:

- "A Drawing Survival Kit" explains how to clear the window and force drawing to be completed. It also gives you basic information about controlling the color of geometric objects and describing a coordinate system.
- "Describing Points, Lines, and Polygons" shows you what the set of primitive geometric objects is and how to draw them.
- "Basic State Management" describes how to turn on and off some states (modes) and query state variables.
- "Displaying Points, Lines, and Polygons" explains what control you have over the details of how primitives are drawn for example, what diameter points have, whether lines are solid or dashed, and whether polygons are outlined or filled.
- "Normal Vectors" discusses how to specify normal vectors for geometric objects and (briefly) what these vectors are for.
- "Vertex Arrays" shows you how to put lots of geometric data into just a few arrays and how, with only a few function calls, to render the geometry it describes. Reducing function calls may increase the efficiency and performance of rendering.
- "Attribute Groups" reveals how to query the current value of state variables and how to save and restore several related state values all at once.
- "Some Hints for Building Polygonal Models of Surfaces" explores the issues and techniques involved in constructing polygonal approximations to surfaces.

One thing to keep in mind as you read the rest of this chapter is that with OpenGL, unless you specify otherwise, every time you issue a drawing command, the specified object is drawn. This might seem obvious, but in some systems, you first make a list of things to draw. When your list is complete, you tell the graphics hardware to draw the items in the list. The first style is called *immediate-mode* graphics and is the default OpenGL style. In addition to using immediate mode, you can choose to save some commands in a list (called a *display list*) for later drawing. Immediate-mode graphics are typically easier to program, but display lists are often more efficient. Chapter 7 tells you how to use display lists and why you might want to use them.

A Drawing Survival Kit

This section explains how to clear the window in preparation for drawing, set the color of objects that are to be drawn, and force drawing to be completed. None of these subjects has anything to do with geometric objects in a direct way, but any program that draws geometric objects has to deal with these

issues.

Clearing the Window

Drawing on a computer screen is different from drawing on paper in that the paper starts out white, and all you have to do is draw the picture. On a computer, the memory holding the picture is usually filled with the last picture you drew, so you typically need to clear it to some background color before you start to draw the new scene. The color you use for the background depends on the application. For a word processor, you might clear to white (the color of the paper) before you begin to draw the text. If you're drawing a view from a spaceship, you clear to the black of space before beginning to draw the stars, planets, and alien spaceships. Sometimes you might not need to clear the screen at all; for example, if the image is the inside of a room, the entire graphics window gets covered as you draw all the walls.

At this point, you might be wondering why we keep talking about *clearing* the window - why not just draw a rectangle of the appropriate color that's large enough to cover the entire window? First, a special command to clear a window can be much more efficient than a general-purpose drawing command. In addition, as you'll see in Chapter 3, OpenGL allows you to set the coordinate system, viewing position, and viewing direction arbitrarily, so it might be difficult to figure out an appropriate size and location for a window-clearing rectangle. Finally, on many machines, the graphics hardware consists of multiple buffers in addition to the buffer containing colors of the pixels that are displayed. These other buffers must be cleared from time to time, and it's convenient to have a single command that can clear any combination of them. (See Chapter 10 for a discussion of all the possible buffers.)

You must also know how the colors of pixels are stored in the graphics hardware known as *bitplanes*. There are two methods of storage. Either the red, green, blue, and alpha (RGBA) values of a pixel can be directly stored in the bitplanes, or a single index value that references a color lookup table is stored. RGBA color-display mode is more commonly used, so most of the examples in this book use it. (See Chapter 4 for more information about both display modes.) You can safely ignore all references to alpha values until Chapter 6.

As an example, these lines of code clear an RGBA mode window to black:

glClearColor(0.0, 0.0, 0.0, 0.0); glClear(GL_COLOR_BUFFER_BIT);

The first line sets the clearing color to black, and the next command clears the entire window to the current clearing color. The single parameter to **glClear()** indicates which buffers are to be cleared. In this case, the program clears only the color buffer, where the image displayed on the screen is kept. Typically, you set the clearing color once, early in your application, and then you clear the buffers as often as necessary. OpenGL keeps track of the current clearing color as a state variable rather than requiring you to specify it each time a buffer is cleared.

Chapter 4 and Chapter 10 talk about how other buffers are used. For now, all you need to know is that clearing them is simple. For example, to clear both the color buffer and the depth buffer, you would use the following sequence of commands:

glClearColor(0.0, 0.0, 0.0, 0.0); glClearDepth(1.0);

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

In this case, the call to **glClearColor**() is the same as before, the **glClearDepth**() command specifies the value to which every pixel of the depth buffer is to be set, and the parameter to the **glClear**() command now consists of the bitwise OR of all the buffers to be cleared. The following summary of **glClear**() includes a table that lists the buffers that can be cleared, their names, and the chapter where each type of buffer is discussed.

void **glClearColor**(GLclampf red, GLclampf green, GLclampf blue,

GLclampf alpha);

Sets the current clearing color for use in clearing color buffers in RGBA mode. (See Chapter 4 for more information on RGBA mode.) The red, green, blue, and alpha values are clamped if necessary to the range [0,1]. The default clearing color is (0, 0, 0, 0), which is black. void **glClear**(GLbitfield mask);

Clears the specified buffers to their current clearing values. The mask argument is a bitwise-ORed combination of the values listed in Table 2-1.

Buffer	Name	Reference
Color buffer	GL_COLOR_BUFFER_BIT	Chapter 4
Depth buffer	GL_DEPTH_BUFFER_BIT	Chapter 10
Accumulation buffer	GL_ACCUM_BUFFER_BIT	Chapter 10
Stencil buffer	GL_STENCIL_BUFFER_BIT	Chapter 10

Table 2-1 : Clearing Buffers

Before issuing a command to clear multiple buffers, you have to set the values to which each buffer is to be cleared if you want something other than the default RGBA color, depth value, accumulation color, and stencil index. In addition to the **glClearColor()** and **glClearDepth()** commands that set the current values for clearing the color and depth buffers, **glClearIndex()**, **glClearAccum()**, and **glClearStencil()** specify the *color index*, accumulation color, and stencil index used to clear the corresponding buffers. (See Chapter 4 and Chapter 10 for descriptions of these buffers and their uses.)

OpenGL allows you to specify multiple buffers because clearing is generally a slow operation, since every pixel in the window (possibly millions) is touched, and some graphics hardware allows sets of buffers to be cleared simultaneously. Hardware that doesn't support simultaneous clears performs them sequentially. The difference between

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

and

```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

is that although both have the same final effect, the first example might run faster on many machines. It certainly won't run more slowly.

Specifying a Color

With OpenGL, the description of the shape of an object being drawn is independent of the description of its color. Whenever a particular geometric object is drawn, it's drawn using the currently specified coloring scheme. The coloring scheme might be as simple as "draw everything in fire-engine red," or might be as complicated as "assume the object is made out of blue plastic, that there's a yellow spotlight pointed in such and such a direction, and that there's a general low-level reddish-brown light everywhere else." In general, an OpenGL programmer first sets the color or coloring scheme and then draws the objects. Until the color or coloring scheme is changed, all objects are drawn in that color or using that coloring scheme. This method helps OpenGL achieve higher drawing performance than would result if it didn't keep track of the current color.

For example, the pseudocode

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

draws objects A and B in red, and object C in blue. The command on the fourth line that sets the current color to green is wasted.

Coloring, lighting, and shading are all large topics with entire chapters or large sections devoted to them. To draw geometric primitives that can be seen, however, you need some basic knowledge of how to set the current color; this information is provided in the next paragraphs. (See Chapter 4 and Chapter 5 for details on these topics.)

To set a color, use the command glColor3f(). It takes three parameters, all of which are floating-point numbers between 0.0 and 1.0. The parameters are, in order, the red, green, and blue *components* of the color. You can think of these three values as specifying a "mix" of colors: 0.0 means don't use any of that component, and 1.0 means use all you can of that component. Thus, the code

glColor3f(1.0, 0.0, 0.0);

makes the brightest red the system can draw, with no green or blue components. All zeros makes black; in contrast, all ones makes white. Setting all three components to 0.5 yields gray (halfway between black and white). Here are eight commands and the colors they would set.

glColor3f(0.0,	0.0,	0.0);	black
glColor3f(1.0,	0.0,	0.0);	red
glColor3f(0.0,	1.0,	0.0);	green
glColor3f(1.0,	1.0,	0.0);	yellow
glColor3f(0.0,	0.0,	1.0);	blue
glColor3f(1.0,	0.0,	1.0);	magenta

glColor3f(0.0,	1.0,	1.0);	cyan
glColor3f(1.0,	1.0,	1.0);	white

You might have noticed earlier that the routine to set the clearing color, **glClearColor**(), takes four parameters, the first three of which match the parameters for **glColor3f**(). The fourth parameter is the alpha value; it's covered in detail in "Blending" in Chapter 6. For now, set the fourth parameter of **glClearColor**() to 0.0, which is its default value.

Forcing Completion of Drawing

As you saw in "OpenGL Rendering Pipeline" in Chapter 1, most modern graphics systems can be thought of as an assembly line. The main central processing unit (CPU) issues a drawing command. Perhaps other hardware does geometric transformations. Clipping is performed, followed by shading and/or texturing. Finally, the values are written into the bitplanes for display. In high-end architectures, each of these operations is performed by a different piece of hardware that's been designed to perform its particular task quickly. In such an architecture, there's no need for the CPU to wait for each drawing command to complete before issuing the next one. While the CPU is sending a vertex down the pipeline, the transformation hardware is working on transforming the last one sent, the one before that is being clipped, and so on. In such a system, if the CPU waited for each command to complete before issuing the next, there could be a huge performance penalty.

In addition, the application might be running on more than one machine. For example, suppose that the main program is running elsewhere (on a machine called the client) and that you're viewing the results of the drawing on your workstation or terminal (the server), which is connected by a network to the client. In that case, it might be horribly inefficient to send each command over the network one at a time, since considerable overhead is often associated with each network transmission. Usually, the client gathers a collection of commands into a single network packet before sending it. Unfortunately, the network code on the client typically has no way of knowing that the graphics program is finished drawing a frame or scene. In the worst case, it waits forever for enough additional drawing commands to fill a packet, and you never see the completed drawing.

For this reason, OpenGL provides the command **glFlush(**), which forces the client to send the network packet even though it might not be full. Where there is no network and all commands are truly executed immediately on the server, **glFlush(**) might have no effect. However, if you're writing a program that you want to work properly both with and without a network, include a call to **glFlush(**) at the end of each frame or scene. Note that **glFlush(**) doesn't wait for the drawing to complete - it just forces the drawing to begin execution, thereby guaranteeing that all previous commands *execute* in finite time even if no further rendering commands are executed.

There are other situations where **glFlush()** is useful.

- Software renderers that build image in system memory and don't want to constantly update the screen.
- Implementations that gather sets of rendering commands to amortize start-up costs. The aforementioned network transmission example is one instance of this.

void glFlush(void);

Forces previously issued OpenGL commands to begin execution, thus guaranteeing that they complete in finite time.

A few commands - for example, commands that swap buffers in double-buffer mode - automatically flush pending commands onto the network before they can occur.

If **glFlush**() isn't sufficient for you, try **glFinish**(). This command flushes the network as **glFlush**() does and then waits for notification from the graphics hardware or network indicating that the drawing is complete in the framebuffer. You might need to use **glFinish**() if you want to synchronize tasks - for example, to make sure that your three-dimensional rendering is on the screen before you use Display PostScript to draw labels on top of the rendering. Another example would be to ensure that the drawing is complete before it begins to accept user input. After you issue a **glFinish**() command, your graphics process is blocked until it receives notification from the graphics hardware that the drawing is complete. Keep in mind that excessive use of **glFinish**() can reduce the performance of your application, especially if you're running over a network, because it requires round-trip communication. If **glFlush**() is sufficient for your needs, use it instead of **glFinish**().

void glFinish(void);

Forces all previously issued OpenGL commands to complete. This command doesn't return until all effects from previous commands are fully realized.

Coordinate System Survival Kit

Whenever you initially open a window or later move or resize that window, the window system will send an event to notify you. If you are using GLUT, the notification is automated; whatever routine has been registered to **glutReshapeFunc()** will be called. You must register a callback function that will

- Reestablish the rectangular region that will be the new rendering canvas
- Define the coordinate system to which objects will be drawn

In Chapter 3 you'll see how to define three-dimensional coordinate systems, but right now, just create a simple, basic two-dimensional coordinate system into which you can draw a few objects. Call **glutReshapeFunc(reshape)**, where **reshape()** is the following function shown in Example 2-1.

Example 2-1 : Reshape Callback Function

```
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
```

The internals of GLUT will pass this function two arguments: the width and height, in pixels, of the new, moved, or resized window. **glViewport()** adjusts the pixel rectangle for drawing to be the entire new window. The next three routines adjust the coordinate system for drawing so that the lower-left corner is (0, 0), and the upper-right corner is (w, h) (See Figure 2-1).

To explain it another way, think about a piece of graphing paper. The w and h values in **reshape**() represent how many columns and rows of squares are on your graph paper. Then you have to put axes on the graph paper. The **gluOrtho2D**() routine puts the origin, (0, 0), all the way in the lowest, leftmost square, and makes each square represent one unit. Now when you render the points, lines, and polygons in the rest of this chapter, they will appear on this paper in easily predictable squares. (For now, keep all your objects two-dimensional.)

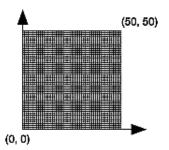


Figure 2-1 : Coordinate System Defined by w = 50, h = 50

Describing Points, Lines, and Polygons

This section explains how to describe OpenGL geometric primitives. All geometric primitives are eventually described in terms of their *vertices* - coordinates that define the points themselves, the endpoints of line segments, or the corners of polygons. The next section discusses how these primitives are displayed and what control you have over their display.

What Are Points, Lines, and Polygons?

You probably have a fairly good idea of what a mathematician means by the terms *point*, line, and polygon. The OpenGL meanings are similar, but not quite the same.

One difference comes from the limitations of computer-based calculations. In any OpenGL implementation, floating-point calculations are of finite precision, and they have round-off errors. Consequently, the coordinates of OpenGL points, lines, and polygons suffer from the same problems.

Another more important difference arises from the limitations of a raster graphics display. On such a display, the smallest displayable unit is a pixel, and although pixels might be less than 1/100 of an inch wide, they are still much larger than the mathematician's concepts of infinitely small (for points) or infinitely thin (for lines). When OpenGL performs calculations, it assumes points are represented as vectors of floating-point numbers. However, a point is typically (but not always) drawn as a single pixel, and many different points with slightly different coordinates could be drawn by OpenGL on the same pixel.

Points

A point is represented by a set of floating-point numbers called a vertex. All internal calculations are done as if vertices are three-dimensional. Vertices specified by the user as two-dimensional (that is, with only x and y coordinates) are assigned a z coordinate equal to zero by OpenGL.

Advanced

OpenGL works in the homogeneous coordinates of three-dimensional projective geometry, so for internal calculations, all vertices are represented with four floating-point coordinates (x, y, z, w). If w is different from zero, these coordinates correspond to the Euclidean three-dimensional point (x/w, y/w, z/w). You can specify the w coordinate in OpenGL commands, but that's rarely done. If the w coordinate isn't specified, it's understood to be 1.0. (See Appendix F for more information about homogeneous coordinate systems.)

Lines

In OpenGL, the term *line* refers to a *line segment*, not the mathematician's version that extends to infinity in both directions. There are easy ways to specify a connected series of line segments, or even a closed, connected series of segments (see Figure 2-2). In all cases, though, the lines constituting the connected series are specified in terms of the vertices at their endpoints.

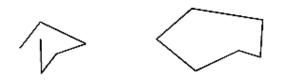


Figure 2-2 : Two Connected Series of Line Segments

Polygons

Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints. Polygons are typically drawn with the pixels in the interior filled in, but you can also draw them as outlines or a set of points. (See "Polygon Details.")

In general, polygons can be complicated, so OpenGL makes some strong restrictions on what constitutes a primitive polygon. First, the edges of OpenGL polygons can't intersect (a mathematician would call a polygon satisfying this condition a *simple polygon*). Second, OpenGL polygons must be *convex*, meaning that they cannot have indentations. Stated precisely, a region is convex if, given any two points in the interior, the line segment joining them is also in the interior. See Figure 2-3 for some examples of valid and invalid polygons. OpenGL, however, doesn't restrict the number of line segments making up the boundary of a convex polygon. Note that polygons with holes can't be described. They are nonconvex, and they can't be drawn with a boundary made up of a single closed loop. Be aware that if you present OpenGL with a nonconvex filled polygon, it might not draw it as you expect. For instance, on most systems no more than the convex hull of the polygon would be filled. On some systems, less than the convex hull might be filled.



Figure 2-3 : Valid and Invalid Polygons

The reason for the OpenGL restrictions on valid polygon types is that it's simpler to provide fast polygon-rendering hardware for that restricted class of polygons. Simple polygons can be rendered quickly. The difficult cases are hard to detect quickly. So for maximum performance, OpenGL crosses its fingers and assumes the polygons are simple.

Many real-world surfaces consist of nonsimple polygons, nonconvex polygons, or polygons with holes. Since all such polygons can be formed from unions of simple convex polygons, some routines to build more complex objects are provided in the GLU library. These routines take complex descriptions and tessellate them, or break them down into groups of the simpler OpenGL polygons that can then be rendered. (See "Polygon Tessellation" in Chapter 11 for more information about the tessellation routines.)

Since OpenGL vertices are always three-dimensional, the points forming the boundary of a particular polygon don't necessarily lie on the same plane in space. (Of course, they do in many cases - if all the *z* coordinates are zero, for example, or if the polygon is a triangle.) If a polygon's vertices don't lie in the same plane, then after various rotations in space, changes in the viewpoint, and projection onto the display screen, the points might no longer form a simple convex polygon. For example, imagine a four-point *quadrilateral* where the points are slightly out of plane, and look at it almost edge-on. You can get a nonsimple polygon that resembles a bow tie, as shown in Figure 2-4, which isn't guaranteed to be rendered correctly. This situation isn't all that unusual if you approximate curved surfaces by quadrilaterals made of points lying on the true surface. You can always avoid the problem by using triangles, since any three points always lie on a plane.

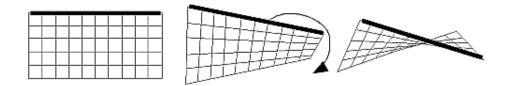


Figure 2-4 : Nonplanar Polygon Transformed to Nonsimple Polygon

Rectangles

Since rectangles are so common in graphics applications, OpenGL provides a filled-rectangle drawing primitive, **glRect***(). You can draw a rectangle as a polygon, as described in "OpenGL Geometric Drawing Primitives," but your particular implementation of OpenGL might have optimized **glRect***() for rectangles.

void glRect{sifd}(TYPEx1, TYPEy1, TYPEx2, TYPEy2); void glRect{sifd}v(TYPE*v1, TYPE*v2);

Draws the rectangle defined by the corner points (x1, y1) and (x2, y2). The rectangle lies in the plane z=0 and has sides parallel to the x- and y-axes. If the vector form of the function is used, the corners are given by two pointers to arrays, each of which contains an (x, y) pair.

Note that although the rectangle begins with a particular orientation in three-dimensional space (in the *x-y* plane and parallel to the axes), you can change this by applying rotations or other transformations. (See Chapter 3 for information about how to do this.)

Curves and Curved Surfaces

Any smoothly curved line or surface can be approximated - to any arbitrary degree of accuracy - by short line segments or small polygonal regions. Thus, subdividing curved lines and surfaces sufficiently and then approximating them with straight line segments or flat polygons makes them appear curved (see Figure 2-5). If you're skeptical that this really works, imagine subdividing until each line segment or polygon is so tiny that it's smaller than a pixel on the screen.

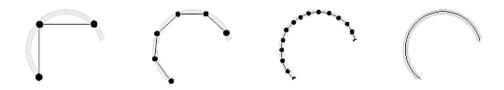


Figure 2-5 : Approximating Curves

Even though curves aren't geometric primitives, OpenGL does provide some direct support for subdividing and drawing them. (See Chapter 12 for information about how to draw curves and curved surfaces.)

Specifying Vertices

With OpenGL, all geometric objects are ultimately described as an ordered set of vertices. You use the **glVertex***() command to specify a vertex.

void glVertex{234}{sifd}[v](TYPEcoords);

Specifies a vertex for use in describing a geometric object. You can supply up to four coordinates (x, y, z, w) for a particular vertex or as few as two (x, y) by selecting the appropriate version of the command. If you use a version that doesn't explicitly specify z or w, z is understood to be 0 and w is understood to be 1. Calls to **glVertex***() are only effective between a **glBegin**() and **glEnd**() pair.

Example 2-2 provides some examples of using glVertex*().

```
Example 2-2 : Legal Uses of glVertex*()
```

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

The first example represents a vertex with three-dimensional coordinates (2, 3, 0). (Remember that if it isn't specified, the *z* coordinate is understood to be 0.) The coordinates in the second example are (0.0, 0.0, 3.1415926535898) (double-precision floating-point numbers). The third example represents the vertex with three-dimensional coordinates (1.15, 0.5, -1.1). (Remember that the *x*, *y*, and *z* coordinates are eventually divided by the *w* coordinate.) In the final example, *dvect* is a pointer to an array of three double-precision floating-point numbers.

On some machines, the vector form of **glVertex***() is more efficient, since only a single parameter needs to be passed to the graphics subsystem. Special hardware might be able to send a whole series of coordinates in a single batch. If your machine is like this, it's to your advantage to arrange your data so that the vertex coordinates are packed sequentially in memory. In this case, there may be some gain in performance by using the vertex array operations of OpenGL. (See "Vertex Arrays.")

OpenGL Geometric Drawing Primitives

Now that you've seen how to specify vertices, you still need to know how to tell OpenGL to create a set of points, a line, or a polygon from those vertices. To do this, you bracket each set of vertices between a call to **glBegin()** and a call to **glEnd()**. The argument passed to **glBegin()** determines what sort of geometric primitive is constructed from the vertices. For example, Example 2-3 specifies the vertices for the polygon shown in Figure 2-6.

Example 2-3 : Filled Polygon

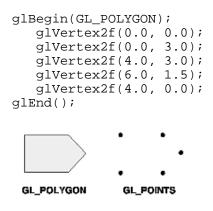


Figure 2-6 : Drawing a Polygon or a Set of Points

If you had used GL_POINTS instead of GL_POLYGON, the primitive would have been simply the five points shown in Figure 2-6. Table 2-2 in the following function summary for **glBegin()** lists the ten possible arguments and the corresponding type of primitive.

void glBegin(GLenum mode);

Marks the beginning of a vertex-data list that describes a geometric primitive. The type of primitive is indicated by mode, which can be any of the values shown in Table 2-2.

Value	Meaning
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon

Table 2-2 : Geometric Primitive Names and Meanings

void glEnd(void);

Marks the end of a vertex-data list.

Figure 2-7 shows examples of all the geometric primitives listed in Table 2-2. The paragraphs that follow the figure describe the pixels that are drawn for each of the objects. Note that in addition to points, several types of lines and polygons are defined. Obviously, you can find many ways to draw the same primitive. The method you choose depends on your vertex data.

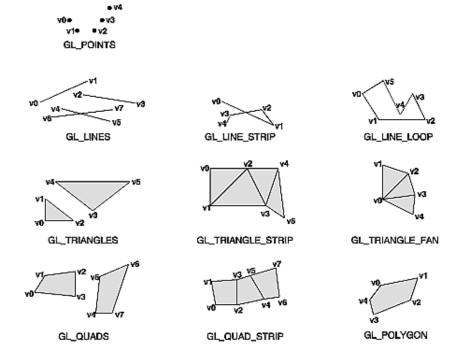


Figure 2-7 : Geometric Primitive Types

As you read the following descriptions, assume that *n* vertices (v0, v1, v2, ..., vn-1) are described between a **glBegin**() and **glEnd**() pair.

GL_POINTS	Draws a point at each of the <i>n</i> vertices.
GL_LINES	Draws a series of unconnected line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If n is odd, the last segment is drawn between vn-3 and vn-2, and vn-1 is ignored.
GL_LINE_STRIP	Draws a line segment from v0 to v1, then from v1 to v2, and so on, finally drawing the segment from vn-2 to vn-1. Thus, a total of n -1 line segments are drawn. Nothing is drawn unless n is larger than 1. There are no restrictions on the vertices describing a line strip (or a line loop); the lines can intersect arbitrarily.
GL_LINE_LOOP	Same as GL_LINE_STRIP, except that a final line segment is drawn from vn-1 to v0, completing a loop.
GL_TRIANGLES	Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v3, v4, v5, and so on. If n isn't an exact multiple of 3, the final one or two vertices are ignored.
GL_TRIANGLE_STRIP	Draws a series of triangles (three-sided polygons) using vertices v0, v1,

	v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. Preserving the orientation is important for some operations, such as culling. (See "Reversing and Culling Polygon Faces") n must be at least 3 for anything to be drawn.
GL_TRIANGLE_FAN	Same as GL_TRIANGLE_STRIP, except that the vertices are v0, v1, v2, then v0, v2, v3, then v0, v3, v4, and so on (see Figure 2-7).
GL_QUADS	Draws a series of quadrilaterals (four-sided polygons) using vertices v0, v1, v2, v3, then v4, v5, v6, v7, and so on. If n isn't a multiple of 4, the final one, two, or three vertices are ignored.
GL_QUAD_STRIP	Draws a series of quadrilaterals (four-sided polygons) beginning with v0, v1, v3, v2, then v2, v3, v5, v4, then v4, v5, v7, v6, and so on (see Figure 2-7). n must be at least 4 before anything is drawn. If n is odd, the final vertex is ignored.
GL_POLYGON	Draws a polygon using the points $v0,, vn-1$ as vertices. <i>n</i> must be at least 3, or nothing is drawn. In addition, the polygon specified must not intersect itself and must be convex. If the vertices don't satisfy these conditions, the results are unpredictable.

Restrictions on Using glBegin() and glEnd()

The most important information about vertices is their coordinates, which are specified by the **glVertex***() command. You can also supply additional vertex-specific data for each vertex - a color, a normal vector, texture coordinates, or any combination of these - using special commands. In addition, a few other commands are valid between a **glBegin**() and **glEnd**() pair. Table 2-3 contains a complete list of such valid commands.

 Table 2-3 : Valid Commands between glBegin() and glEnd()

Command	Purpose of Command	Reference
glVertex*()	set vertex coordinates	Chapter 2
glColor*()	set current color	Chapter 4
glIndex*()	set current color index	Chapter 4
glNormal*()	set normal vector coordinates	Chapter 2
glTexCoord*()	set texture coordinates	Chapter 9
glEdgeFlag*()	control drawing of edges	Chapter 2
glMaterial*()	set material properties	Chapter 5
glArrayElement()	extract vertex array data	Chapter 2
glEvalCoord*(), glEvalPoint*()	generate coordinates	Chapter 12
glCallList(), glCallLists()	execute display list(s)	Chapter 7

No other OpenGL commands are valid between a **glBegin()** and **glEnd()** pair, and making most other OpenGL calls generates an error. Some vertex array commands, such as **glEnableClientState()** and **glVertexPointer()**, when called between **glBegin()** and **glEnd()**, have undefined behavior but do not necessarily generate an error. (Also, routines related to OpenGL, such as **glX*()** routines have undefined behavior between **glBegin()** and **glEnd()**.) These cases should be avoided, and debugging them may be more difficult.

Note, however, that only OpenGL commands are restricted; you can certainly include other programming-language constructs (except for calls, such as the aforementioned **glX***() routines). For example, Example 2-4 draws an outlined circle.

Example 2-4 : Other Constructs between glBegin() and glEnd()

```
#define PI 3.1415926535898
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
for (i = 0; i < circle_points; i++) {
    angle = 2*PI*i/circle_points;
    glVertex2f(cos(angle), sin(angle));
}
glEnd();</pre>
```

Note: This example isn't the most efficient way to draw a circle, especially if you intend to do it

repeatedly. The graphics commands used are typically very fast, but this code calculates an angle and calls the **sin()** and **cos()** routines for each vertex; in addition, there's the loop overhead. (Another way to calculate the vertices of a circle is to use a GLU routine; see "Quadrics: Rendering Spheres, Cylinders, and Disks" in Chapter 11.) If you need to draw lots of circles, calculate the coordinates of the vertices once and save them in an array and create a display list (see Chapter 7), or use vertex arrays to render them.

Unless they are being compiled into a display list, all **glVertex***() commands should appear between some **glBegin**() and **glEnd**() combination. (If they appear elsewhere, they don't accomplish anything.) If they appear in a display list, they are executed only if they appear between a **glBegin**() and a **glEnd**(). (See Chapter 7 for more information about display lists.)

Although many commands are allowed between **glBegin()** and **glEnd()**, vertices are generated only when a **glVertex*()** command is issued. At the moment **glVertex*()** is called, OpenGL assigns the resulting vertex the current color, texture coordinates, normal vector information, and so on. To see this, look at the following code sequence. The first point is drawn in red, and the second and third ones in blue, despite the extra color commands.

```
glBegin(GL_POINTS);
glColor3f(0.0, 1.0, 0.0); /* green */
glColor3f(1.0, 0.0, 0.0); /* red */
glVertex(...);
glColor3f(1.0, 1.0, 0.0); /* yellow */
glColor3f(0.0, 0.0, 1.0); /* blue */
glVertex(...);
glVertex(...);
glEnd();
```

You can use any combination of the 24 versions of the **glVertex***() command between **glBegin**() and **glEnd**(), although in real applications all the calls in any particular instance tend to be of the same form. If your vertex-data specification is consistent and repetitive (for example, **glColor***, **glVertex***, **glColor***, **glVertex***,...), you may enhance your program's performance by using vertex arrays. (See "Vertex Arrays.")

Basic State Management

In the previous section, you saw an example of a state variable, the current RGBA color, and how it can be associated with a primitive. OpenGL maintains many states and state variables. An object may be rendered with lighting, texturing, hidden surface removal, fog, or some other states affecting its appearance.

By default, most of these states are initially inactive. These states may be costly to activate; for example, turning on texture mapping will almost certainly slow down the speed of rendering a primitive. However, the quality of the image will improve and look more realistic, due to the enhanced graphics capabilities.

To turn on and off many of these states, use these two simple commands:

void glEnable(GLenum cap); void glDisable(GLenum cap);

glEnable() turns on a capability, and glDisable() turns it off. There are over 40 enumerated values that can be passed as a parameter to glEnable() or glDisable(). Some examples of these are GL_BLEND (which controls blending RGBA values), GL_DEPTH_TEST (which controls depth comparisons and updates to the depth buffer), GL_FOG (which controls fog), GL_LINE_STIPPLE (patterned lines), GL_LIGHTING (you get the idea), and so forth.

You can also check if a state is currently enabled or disabled.

GLboolean gllsEnabled(GLenum capability)

Returns GL_TRUE or GL_FALSE, depending upon whether the queried capability is currently activated.

The states you have just seen have two settings: on and off. However, most OpenGL routines set values for more complicated state variables. For example, the routine **glColor3f**() sets three values, which are part of the GL_CURRENT_COLOR state. There are five querying routines used to find out what values are set for many states:

void glGetBooleanv(GLenum pname, GLboolean *params); void glGetIntegerv(GLenum pname, GLint *params); void glGetFloatv(GLenum pname, GLfloat *params); void glGetDoublev(GLenum pname, GLdouble *params); void glGetPointerv(GLenum pname, GLvoid **params);

Obtains Boolean, integer, floating-point, double-precision, or pointer state variables. The pname argument is a symbolic constant indicating the state variable to return, and params is a pointer to an array of the indicated type in which to place the returned data. See the tables in Appendix B for the possible values for pname. For example, to get the current RGBA color, a table in Appendix B suggests you use glGetIntegerv(GL_CURRENT_COLOR, params) or glGetFloatv(GL_CURRENT_COLOR, params). A type conversion is performed if necessary to return the desired variable as the requested data type.

These querying routines handle most, but not all, requests for obtaining state information. (See "The Query Commands" in Appendix B for an additional 16 querying routines.)

Displaying Points, Lines, and Polygons

By default, a point is drawn as a single pixel on the screen, a line is drawn solid and one pixel wide, and polygons are drawn solidly filled in. The following paragraphs discuss the details of how to change these default display modes.

Point Details

To control the size of a rendered point, use **glPointSize**() and supply the desired size in pixels as the argument.

void glPointSize(GLfloat size);

Sets the width in pixels for rendered points; size must be greater than 0.0 and by default is 1.0.

The actual collection of pixels on the screen which are drawn for various point widths depends on whether antialiasing is enabled. (Antialiasing is a technique for smoothing points and lines as they're rendered; see "Antialiasing" in Chapter 6 for more detail.) If antialiasing is disabled (the default), fractional widths are rounded to integer widths, and a screen-aligned square region of pixels is drawn. Thus, if the width is 1.0, the square is 1 pixel by 1 pixel; if the width is 2.0, the square is 2 pixels by 2 pixels, and so on.

With antialiasing enabled, a circular *group* of pixels is drawn, and the pixels on the boundaries are typically drawn at less than full intensity to give the edge a smoother appearance. In this mode, non-integer widths aren't rounded.

Most OpenGL implementations support very large point sizes. The maximum size for antialiased points is queryable, but the same information is not available for standard, aliased points. A particular implementation, however, might limit the size of standard, aliased points to not less than its maximum antialiased point size, rounded to the nearest integer value. You can obtain this floating-point value by using GL_POINT_SIZE_RANGE with **glGetFloatv**().

Line Details

With OpenGL, you can specify lines with different widths and lines that are *stippled* in various ways - dotted, dashed, drawn with alternating dots and dashes, and so on.

Wide Lines

void glLineWidth(GLfloat width);

Sets the width in pixels for rendered lines; width must be greater than 0.0 and by default is 1.0.

The actual rendering of lines is affected by the antialiasing mode, in the same way as for points. (See "Antialiasing" in Chapter 6.) Without antialiasing, widths of 1, 2, and 3 draw lines 1, 2, and 3 pixels wide. With antialiasing enabled, non-integer line widths are possible, and pixels on the boundaries are typically drawn at less than full intensity. As with point sizes, a particular OpenGL implementation might limit the width of nonantialiased lines to its maximum antialiased line width, rounded to the nearest integer value. You can obtain this floating-point value by using GL_LINE_WIDTH_RANGE with **glGetFloatv**().

Note: Keep in mind that by default lines are 1 pixel wide, so they appear wider on lower-resolution screens. For computer displays, this isn't typically an issue, but if you're using OpenGL to render to a high-resolution plotter, 1-pixel lines might be nearly invisible. To obtain resolution-independent line widths, you need to take into account the physical dimensions of pixels.

Advanced

With nonantialiased wide lines, the line width isn't measured perpendicular to the line. Instead, it's measured in the y direction if the absolute value of the slope is less than 1.0; otherwise, it's measured in the x direction. The rendering of an antialiased line is exactly equivalent to the rendering of a filled

rectangle of the given width, centered on the exact line.

Stippled Lines

To make stippled (dotted or dashed) lines, you use the command **glLineStipple()** to define the stipple pattern, and then you enable line stippling with **glEnable()**.

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
```

void glLineStipple(GLint factor, GLushort pattern);

Sets the current stippling pattern for lines. The pattern argument is a 16-bit series of 0s and 1s, and it's repeated as necessary to stipple a given line. A 1 indicates that drawing occurs, and 0 that it does not, on a pixel-by-pixel basis, beginning with the low-order bit of the pattern. The pattern can be stretched out by using factor, which multiplies each subseries of consecutive 1s and 0s. Thus, if three consecutive 1s appear in the pattern, they're stretched to six if factor is 2. factor is clamped to lie between 1 and 255. Line stippling must be enabled by passing GL_LINE_STIPPLE to glEnable(); it's disabled by passing the same argument to glDisable().

With the preceding example and the pattern 0x3F07 (which translates to 0011111100000111 in binary), a line would be drawn with 3 pixels on, then 5 off, 6 on, and 2 off. (If this seems backward, remember that the low-order bit is used first.) If *factor* had been 2, the pattern would have been elongated: 6 pixels on, 10 off, 12 on, and 4 off. Figure 2-8 shows lines drawn with different patterns and repeat factors. If you don't enable line stippling, drawing proceeds as if *pattern* were 0xFFFF and *factor* 1. (Use **glDisable**() with GL_LINE_STIPPLE to disable stippling.) Note that stippling can be used in combination with wide lines to produce wide stippled lines.

PATTERN	FACTOR
0x00FF	1
0x00FF	2
0x0C0F	1
0x0C0F	3
OXAAAA	1
0xAAAA	2
0xAAAA	3
0xAAAA	4

Figure 2-8 : Stippled Lines

One way to think of the stippling is that as the line is being drawn, the pattern is shifted by 1 bit each time a pixel is drawn (or *factor* pixels are drawn, if *factor* isn't 1). When a series of connected line segments is drawn between a single **glBegin**() and **glEnd**(), the pattern continues to shift as one segment turns into the next. This way, a stippling pattern continues across a series of connected line segments. When **glEnd**() is executed, the pattern is reset, and - if more lines are drawn before stippling is disabled - the stippling restarts at the beginning of the pattern. If you're drawing lines with GL_LINES, the pattern resets for each independent line.

Example 2-5 illustrates the results of drawing with a couple of different stipple patterns and line widths. It also illustrates what happens if the lines are drawn as a series of individual segments instead of a

single connected line strip. The results of running the program appear in Figure 2-9.

Figure 2-9 : Wide Stippled Lines

Example 2-5 : Line Stipple Patterns: lines.c

```
#include <GL/gl.h>
#include <GL/glut.h>
#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
   glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}
void display(void)
{
   int i;
   glClear (GL_COLOR_BUFFER_BIT);
/* select white for all lines */
   glColor3f (1.0, 1.0, 1.0);
/* in 1st row, 3 lines, each with a different stipple */
   glEnable (GL LINE STIPPLE);
   glLineStipple (1, 0x0101); /* dotted */
   drawOneLine (50.0, 125.0, 150.0, 125.0);
   glLineStipple (1, 0x00FF); /* dashed */
drawOneLine (150.0, 125.0, 250.0, 125.0);
   glLineStipple (1, 0x1C47); /* dash/dot/dash */
drawOneLine (250.0, 125.0, 350.0, 125.0);
/* in 2nd row, 3 wide lines, each with different stipple */
   glLineWidth (5.0);
   glLineStipple (1, 0x0101); /* dotted */
   drawOneLine (50.0, 100.0, 150.0, 100.0);
   glLineStipple (1, 0x00FF); /* dashed */
   drawOneLine (150.0, 100.0, 250.0, 100.0);
   glLineStipple (1, 0x1C47); /* dash/dot/dash */
   drawOneLine (250.0, 100.0, 350.0, 100.0);
   glLineWidth (1.0);
/* in 3rd row, 6 lines, with dash/dot/dash stipple
                                                       */
/* as part of a single connected line strip
                                                        */
   glLineStipple (1, 0x1C47); /* dash/dot/dash */
```

```
glBegin (GL_LINE_STRIP);
   for (i = 0; i < 7; i++)
      glVertex2f (50.0 + ((GLfloat) i * 50.0), 75.0);
   glEnd ();
/* in 4th row, 6 independent lines with same stipple */
   for (i = 0; i < 6; i++)
      drawOneLine (50.0 + ((GLfloat) i * 50.0), 50.0,
         50.0 + ((GLfloat)(i+1) * 50.0), 50.0);
   }
/* in 5th row, 1 line, with dash/dot/dash stipple
                                                      */
/* and a stipple repeat factor of 5
                                                      */
   glLineStipple (5, 0x1C47); /* dash/dot/dash */
   drawOneLine (50.0, 25.0, 350.0, 25.0);
  glDisable (GL LINE STIPPLE);
  glFlush ();
}
void reshape (int w, int h)
{
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (400, 150);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

Polygon Details

Polygons are typically drawn by filling in all the pixels enclosed within the boundary, but you can also draw them as outlined polygons or simply as points at the vertices. A filled polygon might be solidly filled or stippled with a certain pattern. Although the exact details are omitted here, filled polygons are drawn in such a way that if adjacent polygons share an edge or vertex, the pixels making up the edge or vertex are drawn exactly once - they're included in only one of the polygons. This is done so that partially transparent polygons don't have their edges drawn twice, which would make those edges appear darker (or brighter, depending on what color you're drawing with). Note that it might result in narrow polygons having no filled pixels in one or more rows or columns of pixels. Antialiasing polygons is more complicated than for points and lines. (See "Antialiasing" in Chapter 6 for details.)

Polygons as Points, Outlines, or Solids

A polygon has two sides - front and back - and might be rendered differently depending on which side is facing the viewer. This allows you to have cutaway views of solid objects in which there is an obvious

distinction between the parts that are inside and those that are outside. By default, both front and back faces are drawn in the same way. To change this, or to draw only outlines or vertices, use **glPolygonMode**().

void glPolygonMode(GLenum face, GLenum mode);

Controls the drawing mode for a polygon's front and back faces. The parameter face can be GL_FRONT_AND_BACK, GL_FRONT, or GL_BACK; mode can be GL_POINT, GL_LINE, or GL_FILL to indicate whether the polygon should be drawn as points, outlined, or filled. By default, both the front and back faces are drawn filled.

For example, you can have the front faces filled and the back faces outlined with two calls to this routine:

glPolygonMode(GL_FRONT, GL_FILL); glPolygonMode(GL_BACK, GL_LINE);

Reversing and Culling Polygon Faces

By convention, polygons whose vertices appear in counterclockwise order on the screen are called front-facing. You can construct the surface of any "reasonable" solid - a mathematician would call such a surface an orientable manifold (spheres, donuts, and teapots are orientable; Klein bottles and Möbius strips aren't) - from polygons of consistent orientation. In other words, you can use all clockwise polygons, or all counterclockwise polygons. (This is essentially the mathematical definition of *orientable*.)

Suppose you've consistently described a model of an orientable surface but that you happen to have the clockwise orientation on the outside. You can swap what OpenGL considers the back face by using the function **glFrontFace**(), supplying the desired orientation for front-facing polygons.

void glFrontFace(GLenum mode);

Controls how front-facing polygons are determined. By default, mode is GL_CCW, which corresponds to a counterclockwise orientation of the ordered vertices of a projected polygon in window coordinates. If mode is GL_CW, faces with a clockwise orientation are considered front-facing.

In a completely enclosed surface constructed from opaque polygons with a consistent orientation, none of the back-facing polygons are ever visible - they're always obscured by the front-facing polygons. If you are outside this surface, you might enable culling to discard polygons that OpenGL determines are back-facing. Similarly, if you are inside the object, only back-facing polygons are visible. To instruct OpenGL to discard front- or back-facing polygons, use the command **glCullFace(**) and enable culling with **glEnable(**).

void glCullFace(GLenum mode);

Indicates which polygons should be discarded (culled) before they're converted to screen coordinates. The mode is either GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate front-facing, back-facing, or all polygons. To take effect, culling must be enabled using **glEnable**() with GL_CULL_FACE; it can be disabled with **glDisable**() and the same argument.

Advanced

In more technical terms, the decision of whether a face of a polygon is front- or back-facing depends on the sign of the polygon's area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2} \sum_{\substack{i=0\\i=0}}^{n-1} x_i y_{i+1,\overline{0}} x_{i+1} y_i$$

where xi and yi are the x and y window coordinates of the *i*th vertex of the *n*-vertex polygon and

 $i\oplus 1$ is $(i+1) \mod n$.

Assuming that GL_CCW has been specified, if a>0, the polygon corresponding to that vertex is considered to be front-facing; otherwise, it's back-facing. If GL_CW is specified and if a<0, then the corresponding polygon is front-facing; otherwise, it's back-facing.

Try This

Modify Example 2-5 by adding some filled polygons. Experiment with different colors. Try different polygon modes. Also enable culling to see its effect.

Stippling Polygons

By default, filled polygons are drawn with a solid pattern. They can also be filled with a 32-bit by 32-bit window-aligned stipple pattern, which you specify with **glPolygonStipple**().

void glPolygonStipple(const GLubyte *mask);

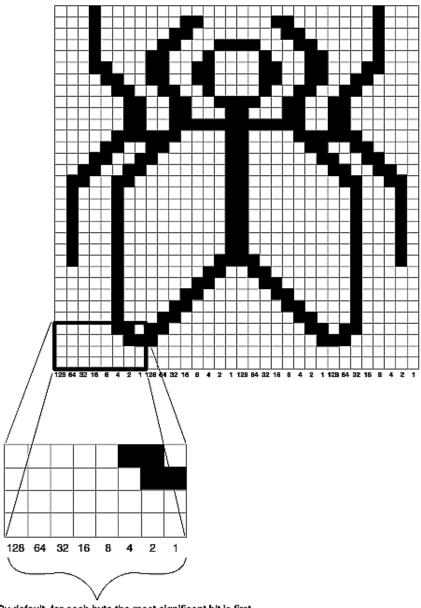
Defines the current stipple pattern for filled polygons. The argument mask is a pointer to a 32 × 32 bitmap that's interpreted as a mask of 0s and 1s. Where a 1 appears, the corresponding pixel in the polygon is drawn, and where a 0 appears, nothing is drawn. Figure 2-10 shows how a stipple pattern is constructed from the characters in mask. Polygon stippling is enabled and disabled by using **glEnable()** and **glDisable()** with GL_POLYGON_STIPPLE as the argument. The interpretation of the mask data is affected by the **glPixelStore*()** GL_UNPACK* modes. (See "Controlling Pixel-Storage Modes" in Chapter 8.)

In addition to defining the current polygon stippling pattern, you must enable stippling:

glEnable(GL_POLYGON_STIPPLE);

Use glDisable() with the same argument to disable polygon stippling.

Figure 2-11 shows the results of polygons drawn unstippled and then with two different stippling patterns. The program is shown in Example 2-6. The reversal of white to black (from Figure 2-10 to Figure 2-11) occurs because the program draws in white over a black background, using the pattern in Figure 2-10 as a stencil.



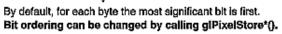


Figure 2-10 : Constructing a Polygon Stipple Pattern

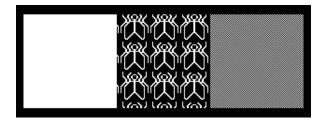


Figure 2-11 : Stippled Polygons

Example 2-6 : Polygon Stipple Patterns: polys.c

```
#include <GL/gl.h>
#include <GL/glut.h>
void display(void)
   GLubyte fly[] = \{
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
      0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
      0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
      0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
      0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
      0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
      0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};
   GLubyte halftone[] = {
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55};
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
/* draw one solid, unstippled rectangle,
                                                 * /
/* then two stippled rectangles
                                                 * /
   glRectf (25.0, 25.0, 125.0, 125.0);
   glEnable (GL_POLYGON_STIPPLE);
   glPolygonStipple (fly);
   glRectf (125.0, 25.0, 225.0, 125.0);
   glPolygonStipple (halftone);
   glRectf (225.0, 25.0, 325.0, 125.0);
   glDisable (GL_POLYGON_STIPPLE);
   glFlush ();
}
void init (void)
```

```
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL FLAT);
}
void reshape (int w, int h)
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL PROJECTION);
  glLoadIdentity ();
  gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT SINGLE | GLUT RGB);
  glutInitWindowSize (350, 150);
  glutCreateWindow (argv[0]);
   init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

You might want to use display lists to store polygon stipple patterns to maximize efficiency. (See "Display-List Design Philosophy" in Chapter 7.)

Marking Polygon Boundary Edges

Advanced

OpenGL can render only convex polygons, but many nonconvex polygons arise in practice. To draw these nonconvex polygons, you typically subdivide them into convex polygons - usually triangles, as shown in Figure 2-12 - and then draw the triangles. Unfortunately, if you decompose a general polygon into triangles and draw the triangles, you can't really use **glPolygonMode**() to draw the polygon's outline, since you get all the triangle outlines inside it. To solve this problem, you can tell OpenGL whether a particular vertex precedes a boundary edge; OpenGL keeps track of this information by passing along with each vertex a bit indicating whether that vertex is followed by a boundary edge. Then, when a polygon is drawn in GL_LINE mode, the nonboundary edges aren't drawn. In Figure 2-12, the dashed lines represent added edges.

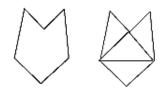


Figure 2-12 : Subdividing a Nonconvex Polygon

By default, all vertices are marked as preceding a boundary edge, but you can manually control the

setting of the edge flag with the command **glEdgeFlag*()**. This command is used between **glBegin()** and **glEnd()** pairs, and it affects all the vertices specified after it until the next **glEdgeFlag()** call is made. It applies only to vertices specified for polygons, triangles, and quads, not to those specified for strips of triangles or quads.

void **glEdgeFlag**(GLboolean flag);

void glEdgeFlagv(const GLboolean *flag);

Indicates whether a vertex should be considered as initializing a boundary edge of a polygon. If flag is GL_TRUE, the edge flag is set to TRUE (the default), and any vertices created are considered to precede boundary edges until this function is called again with flag being GL_FALSE.

As an example, Example 2-7 draws the outline shown in Figure 2-13.

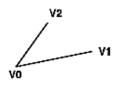


Figure 2-13 : Outlined Polygon Drawn Using Edge Flags

Example 2-7 : Marking Polygon Boundary Edges

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V0);
    glEdgeFlag(GL_FALSE);
    glVertex3fv(V1);
    glEdgeFlag(GL_TRUE);
    glVertex3fv(V2);
glEnd();
```

Normal Vectors

A *normal vector* (or normal, for short) is a vector that points in a direction that's perpendicular to a surface. For a flat surface, one perpendicular direction is the same for every point on the surface, but for a general curved surface, the normal direction might be different at each point on the surface. With OpenGL, you can specify a normal for each polygon or for each vertex. Vertices of the same polygon might share the same normal (for a flat surface) or have different normals (for a curved surface). But you can't assign normals anywhere other than at the vertices.

An object's normal vectors define the orientation of its surface in space - in particular, its orientation relative to light sources. These vectors are used by OpenGL to determine how much light the object receives at its vertices. Lighting - a large topic by itself - is the subject of Chapter 5, and you might want

to review the following information after you've read that chapter. Normal vectors are discussed briefly here because you define normal vectors for an object at the same time you define the object's geometry.

You use **glNormal***() to set the current normal to the value of the argument passed in. Subsequent calls to **glVertex***() cause the specified vertices to be assigned the current normal. Often, each vertex has a different normal, which necessitates a series of alternating calls, as in Example 2-8.

Example 2-8 : Surface Normals at Vertices

```
glBegin (GL_POLYGON);
glNormal3fv(n0);
glVertex3fv(v0);
glNormal3fv(n1);
glVertex3fv(v1);
glNormal3fv(n2);
glVertex3fv(v2);
glNormal3fv(n3);
glVertex3fv(v3);
glEnd();
```

void glNormal3{bsidf}(TYPEnx, TYPEny, TYPEnz); void glNormal3{bsidf}v(const TYPE *v);

> Sets the current normal vector as specified by the arguments. The nonvector version (without the v) takes three arguments, which specify an (nx, ny, nz) vector that's taken to be the normal. Alternatively, you can use the vector version of this function (with the v) and supply a single array of three elements to specify the desired normal. The b, s, and i versions scale their parameter values linearly to the range [-1.0,1.0].

There's no magic to finding the normals for an object - most likely, you have to perform some calculations that might include taking derivatives - but there are several techniques and tricks you can use to achieve certain effects. Appendix E explains how to find normal vectors for surfaces. If you already know how to do this, if you can count on always being supplied with normal vectors, or if you don't want to use the lighting facility provided by OpenGL lighting facility, you don't need to read this appendix.

Note that at a given point on a surface, two vectors are perpendicular to the surface, and they point in opposite directions. By convention, the normal is the one that points to the outside of the surface being modeled. (If you get inside and outside reversed in your model, just change every normal vector from (x, y, z) to (- &xgr; , -y, -z)).

Also, keep in mind that since normal vectors indicate direction only, their length is mostly irrelevant. You can specify normals of any length, but eventually they have to be converted to having a length of 1 before lighting calculations are performed. (A vector that has a length of 1 is said to be of unit length, or normalized.) In general, you should supply normalized normal vectors. To make a normal vector of unit length, divide each of its x, y, z components by the length of the normal:

Length = $\sqrt{x^2 + y^2 + z^2}$

Normal vectors remain normalized as long as your model transformations include only rotations and

translations. (See Chapter 3 for a discussion of transformations.) If you perform irregular transformations (such as scaling or multiplying by a shear matrix), or if you specify nonunit-length normals, then you should have OpenGL automatically normalize your normal vectors after the transformations. To do this, call **glEnable**() with GL_NORMALIZE as its argument. By default, automatic normalization is disabled. Note that automatic normalization typically requires additional calculations that might reduce the performance of your application.

Vertex Arrays

You may have noticed that OpenGL requires many function calls to render geometric primitives. Drawing a 20-sided polygon requires 22 function calls: one call to **glBegin**(), one call for each of the vertices, and a final call to **glEnd**(). In the two previous code examples, additional information (polygon boundary edge flags or surface normals) added function calls for each vertex. This can quickly double or triple the number of function calls required for one geometric object. For some systems, function calls have a great deal of overhead and can hinder performance.

An additional problem is the redundant processing of vertices that are shared between adjacent polygons. For example, the cube in Figure 2-14 has six faces and eight shared vertices. Unfortunately, using the standard method of describing this object, each vertex would have to be specified three times: once for every face that uses it. So 24 vertices would be processed, even though eight would be enough.



Figure 2-14 : Six Sides; Eight Shared Vertices

OpenGL has vertex array routines that allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls. Using vertex array routines, all 20 vertices in a 20-sided polygon could be put into one array and called with one function. If each vertex also had a surface normal, all 20 surface normals could be put into another array and also called with one function.

Arranging data in vertex arrays may increase the performance of your application. Using vertex arrays reduces the number of function calls, which improves performance. Also, using vertex arrays may allow non-redundant processing of shared vertices. (Vertex sharing is not supported on all implementations of OpenGL.)

Note: Vertex arrays are standard in version 1.1 of OpenGL but were not part of the OpenGL 1.0 specification. With OpenGL 1.0, some vendors have implemented vertex arrays as an extension.

There are three steps to using vertex arrays to render geometry.

1. Activate (enable) up to six arrays, each to store a different type of data: vertex coordinates, RGBA

colors, color indices, surface normals, texture coordinates, or polygon edge flags.

- 2. Put data into the array or arrays. The arrays are accessed by the addresses of (that is, pointers to) their memory locations. In the client-server model, this data is stored in the client's address space.
- 3. Draw geometry with the data. OpenGL obtains the data from all activated arrays by dereferencing the pointers. In the client-server model, the data is transferred to the server's address space. There are three ways to do this:
 - 1. Accessing individual array elements (randomly hopping around)
 - 2. Creating a list of individual array elements (methodically hopping around)
 - 3. Processing sequential array elements

The dereferencing method you choose may depend upon the type of problem you encounter.

Interleaved vertex array data is another common method of organization. Instead of having up to six different arrays, each maintaining a different type of data (color, surface normal, coordinate, and so on), you might have the different types of data mixed into a single array. (See "Interleaved Arrays" for two methods of solving this.)

Step 1: Enabling Arrays

The first step is to call **glEnableClientState()** with an enumerated parameter, which activates the chosen array. In theory, you may need to call this up to six times to activate the six available arrays. In practice, you'll probably activate only between one to four arrays. For example, it is unlikely that you would activate both GL_COLOR_ARRAY and GL_INDEX_ARRAY, since your program's display mode supports either RGBA mode or color-index mode, but probably not both simultaneously.

void glEnableClientState(GLenum array)

Specifies the array to enable. Symbolic constants GL_VERTEX_ARRAY, GL_COLOR_ARRAY, GL_INDEX_ARRAY, GL_NORMAL_ARRAY, GL_TEXTURE_COORD_ARRAY, and GL_EDGE_FLAG_ARRAY are acceptable parameters.

If you use lighting, you may want to define a surface normal for every vertex. (See "Normal Vectors.") To use vertex arrays for that case, you activate both the surface normal and vertex coordinate arrays:

```
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Suppose that you want to turn off lighting at some point and just draw the geometry using a single color. You want to call **glDisable**() to turn off lighting states (see Chapter 5). Now that lighting has been deactivated, you also want to stop changing the values of the surface normal state, which is wasted effort. To do that, you call

```
glDisableClientState(GL_NORMAL_ARRAY);
```

void glDisableClientState(GLenum array);

Specifies the array to disable. Accepts the same symbolic constants as glEnableClientState().

You might be asking yourself why the architects of OpenGL created these new (and long!) command names, **gl*ClientState**(). Why can't you just call **glEnable**() and **glDisable**()? One reason is that **glEnable**() and **glDisable**() can be stored in a display list, but the specification of vertex arrays cannot, because the data remains on the client's side.

Step 2: Specifying Data for the Arrays

There is a straightforward way by which a single command specifies a single array in the client space. There are six different routines to specify arrays - one routine for each kind of array. There is also a command that can specify several client-space arrays at once, all originating from a single interleaved array.

void glVertexPointer(GLint size, GLenum type, GLsizei stride,

const GLvoid *pointer);

Specifies where spatial coordinate data can be accessed. pointer is the memory address of the first coordinate of the first vertex in the array. type specifies the data type (GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE) of each coordinate in the array. size is the number of coordinates per vertex, which must be 2, 3, or 4. stride is the byte offset between consecutive vertexes. If stride is 0, the vertices are understood to be tightly packed in the array.

To access the other five arrays, there are five similar routines:

void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer); void glIndexPointer(GLenum type, GLsizei stride, const GLvoid *pointer); void glNormalPointer(GLenum type, GLsizei stride, const GLvoid *pointer); void glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const GLvoid *pointer); void glEdgeFlagPointer(GLsizei stride, const GLvoid *pointer);

The main differences among the routines are whether size and type are unique or must be specified. For example, a surface normal always has three components, so it is redundant to specify its size. An edge flag is always a single Boolean, so neither size nor type needs to be mentioned. Table 2-4 displays legal values for size and data types.

Table 2-4 : Vertex Array Sizes (Values per Vertex) and Data Types(continued)

Command	Sizes	Values for <i>type</i> Argument
glVertexPointer	2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glNormalPointer	3	GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glColorPointer	3, 4	GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE
glIndexPointer	1	GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glTexCoordPointer	1, 2, 3, 4	GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
glEdgeFlagPointer	1	no type argument (type of data must be GLboolean)

Example 2-9 uses vertex arrays for both RGBA colors and vertex coordinates. RGB floating-point values and their corresponding (x, y) integer coordinates are loaded into the GL_COLOR_ARRAY and GL_VERTEX_ARRAY.

Example 2-9 : Enabling and Loading Vertex Arrays: varray.c

Stride

With a stride of zero, each type of vertex array (RGB color, color index, vertex coordinate, and so on) must be tightly packed. The data in the array must be homogeneous; that is, the data must be all RGB color values, all vertex coordinates, or all some other data similar in some fashion.

Using a stride of other than zero can be useful, especially when dealing with interleaved arrays. In the following array of GL floats, there are six vertices. For each vertex, there are three RGB color values, which alternate with the (x, y, z) vertex coordinates.

static GLfloat intertwined[] =
 {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
 1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
 1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
 0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
 0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
 0.2, 0.2, 1.0, 200.0, 100.0, 0.0];

Stride allows a vertex array to access its desired data at regular intervals in the array. For example, to reference only the color values in the *intertwined* array, the following call starts from the beginning of the array (which could also be passed as *&intertwined[0]*) and jumps ahead 6 * **sizeof**(GLfloat) bytes, which is the size of both the color and vertex coordinate values. This jump is enough to get to the beginning of the data for the next vertex.

glColorPointer (3, GL_FLOAT, 6 * sizeof(GLfloat), intertwined);

For the vertex coordinate pointer, you need to start from further in the array, at the fourth element of *intertwined* (remember that C programmers start counting at zero).

glVertexPointer(3, GL_FLOAT,6*sizeof(GLfloat), &intertwined[3]);

Step 3: Dereferencing and Rendering

Until the contents of the vertex arrays are dereferenced, the arrays remain on the client side, and their contents are easily changed. In Step 3, contents of the arrays are obtained, sent down to the server, and then sent down the graphics processing pipeline for rendering.

There are three ways to obtain data: from a single array element (indexed location), from a sequence of array elements, and from an ordered list of array elements.

Dereference a Single Array Element

void glArrayElement(GLint ith)

Obtains the data of one (the ith) vertex for all currently enabled arrays. For the vertex coordinate array, the corresponding command would be **glVertex**[size][type]**v**(), where size is one of [2,3,4], and type is one of [s,i,f,d] for GLshort, GLint, GLfloat, and GLdouble respectively. Both size and type were defined by **glVertexPointer**(). For other enabled arrays, **glArrayElement**() calls **glEdgeFlagv**(), **glTexCoord**[size][type]**v**(), **glColor**[size][type]**v**(), **glIndex**[type]**v**(), and **glNormal**[type]**v**(). If the vertex coordinate array is enabled, the **glVertex*v**() routine is executed last, after the execution (if enabled) of up to five corresponding array values.

glArrayElement() is usually called between glBegin() and glEnd(). (If called outside,

glArrayElement() sets the current state for all enabled arrays, except for vertex, which has no current state.) In Example 2-10, a triangle is drawn using the third, fourth, and sixth vertices from enabled vertex arrays (again, remember that C programmers begin counting array locations with zero).

Example 2-10 : Using glArrayElement() to Define Colors and Vertices

glEnableClientState (GL_COLOR_ARRAY); glEnableClientState (GL_VERTEX_ARRAY); glColorPointer (3, GL_FLOAT, 0, colors); glVertexPointer (2, GL_INT, 0, vertices); glBegin(GL_TRIANGLES); glArrayElement (2); glArrayElement (3); glArrayElement (5); glEnd();

When executed, the latter five lines of code has the same effect as

```
glBegin(GL_TRIANGLES);
glColor3fv(colors+(2*3*sizeof(GLfloat));
glVertex3fv(vertices+(2*2*sizeof(GLint));
glColor3fv(colors+(3*3*sizeof(GLfloat));
glVertex3fv(vertices+(3*2*sizeof(GLint));
glColor3fv(colors+(5*3*sizeof(GLfloat));
glVertex3fv(vertices+(5*2*sizeof(GLint));
glEnd();
```

Since **glArrayElement**() is only a single function call per vertex, it may reduce the number of function calls, which increases overall performance.

Be warned that if the contents of the array are changed between **glBegin**() and **glEnd**(), there is no guarantee that you will receive original data or changed data for your requested element. To be safe, don't change the contents of any array element which might be accessed until the primitive is completed.

Dereference a List of Array Elements

glArrayElement() is good for randomly "hopping around" your data arrays. A similar routine, **glDrawElements()**, is good for hopping around your data arrays in a more orderly manner.

```
void glDrawElements(GLenum mode, GLsizei count, GLenum type, void *indices):
```

Defines a sequence of geometric primitives using count number of elements, whose indices are stored in the array indices. type must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT, indicating the data type of the indices array. mode specifies what kind of primitives are constructed and is one of the same values that is accepted by **glBegin**(); for example, GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS, and so on.

The effect of **glDrawElements**() is almost the same as this command sequence:

```
int i;
glBegin (mode);
for (i = 0; i < count; i++)
glArrayElement(indices[i]);
glEnd();
```

glDrawElements() additionally checks to make sure *mode*, *count*, and *type* are valid. Also, unlike the preceding sequence, executing **glDrawElements**() leaves several states indeterminate. After execution of **glDrawElements**(), current RGB color, color index, normal coordinates, texture coordinates, and edge flag are indeterminate if the corresponding array has been enabled.

With **glDrawElements**(), the vertices for each face of the cube can be placed in an array of indices. Example 2-11 shows two ways to use **glDrawElements**() to render the cube. Figure 2-15 shows the numbering of the vertices used in Example 2-11.

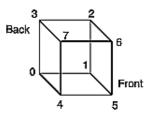


Figure 2-15 : Cube with Numbered Vertices

Example 2-11 : Two Ways to Use glDrawElements()

```
static GLubyte frontIndices = {4, 5, 6, 7};
static GLubyte rightIndices = {1, 2, 6, 5};
static GLubyte bottomIndices = {0, 1, 5, 4};
static GLubyte backIndices = {0, 3, 2, 1};
static GLubyte leftIndices = {0, 4, 7, 3};
static GLubyte topIndices = {2, 3, 7, 6};
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, frontIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, rightIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottomIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, leftIndices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, topIndices);
```

Or better still, crunch all the indices together:

```
static GLubyte allIndices = {4, 5, 6, 7, 1, 2, 6, 5,
0, 1, 5, 4, 0, 3, 2, 1,
0, 4, 7, 3, 2, 3, 7, 6};
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, allIndices);
```

Note: It is an error to encapsulate glDrawElements() between a glBegin()/glEnd() pair.

With both **glArrayElement()** and **glDrawElements()**, it is also possible that your OpenGL implementation caches recently processed vertices, allowing your application to "share" or "reuse" vertices. Take the aforementioned cube, for example, which has six faces (polygons) but only eight vertices. Each vertex is used by exactly three faces. Without **glArrayElement()** or **glDrawElements()**, rendering all six faces would require processing twenty-four vertices, even though sixteen vertices would be redundant. Your implementation of OpenGL may be able to minimize redundancy and process

as few as eight vertices. (Reuse of vertices may be limited to all vertices within a single **glDrawElements**() call or, for **glArrayElement**(), within one **glBegin**()/**glEnd**() pair.)

Dereference a Sequence of Array Elements

While **glArrayElement**() and **glDrawElements**() "hop around" your data arrays, **glDrawArrays**() plows straight through them.

void glDrawArrays(GLenum mode, GLint first, GLsizei count);

Constructs a sequence of geometric primitives using array elements starting at first and ending at first+count-1 of each enabled array. mode specifies what kinds of primitives are constructed and is one of the same values accepted by **glBegin()**; for example, GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS, and so on.

The effect of glDrawArrays() is almost the same as this command sequence:

```
int i;
glBegin (mode);
for (i = 0; i < count; i++)
glArrayElement(first + i);
glEnd();
```

As is the case with **glDrawElements()**, **glDrawArrays()** also performs error checking on its parameter values and leaves the current RGB color, color index, normal coordinates, texture coordinates, and edge flag with indeterminate values if the corresponding array has been enabled.

Try This

• Change the icosahedron drawing routine in Example 2-13 to use vertex arrays.

Interleaved Arrays

Advanced

Earlier in this chapter (in "Stride"), the special case of interleaved arrays was examined. In that section, the array *intertwined*, which interleaves RGB color and 3D vertex coordinates, was accessed by calls to **glColorPointer**() and **glVertexPointer**(). Careful use of stride helped properly specify the arrays.

```
static GLfloat intertwined[] =
    {1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
    1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
    1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
    0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
    0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
    0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

There is also a behemoth routine, **glInterleavedArrays**(), that can specify several vertex arrays at once. **glInterleavedArrays**() also enables and disables the appropriate arrays (so it combines both Steps 1 and 2). The array *intertwined* exactly fits one of the fourteen data interleaving configurations supported by **glInterleavedArrays**(). So to specify the contents of the array *intertwined* into the RGB color and

vertex arrays and enable both arrays, call

glInterleavedArrays (GL_C3F_V3F, 0, intertwined);

This call to **glInterleavedArrays**() enables the GL_COLOR_ARRAY and GL_VERTEX_ARRAY arrays. It disables the GL_INDEX_ARRAY, GL_TEXTURE_COORD_ARRAY, GL_NORMAL_ARRAY, and GL_EDGE_FLAG_ARRAY.

This call also has the same effect as calling **glColorPointer**() and **glVertexPointer**() to specify the values for six vertices into each array. Now you are ready for Step 3: Calling **glArrayElement**(), **glDrawElements**(), or **glDrawArrays**() to dereference array elements.

void **glInterleavedArrays**(GLenum format, GLsizei stride, void *pointer)

Initializes all six arrays, disabling arrays that are not specified in format, and enabling the arrays that are specified. format is one of 14 symbolic constants, which represent 14 data configurations; Table 2-5 displays format values. stride specifies the byte offset between consecutive vertexes. If stride is 0, the vertexes are understood to be tightly packed in the array. pointer is the memory address of the first coordinate of the first vertex in the array.

Note that **glInterleavedArrays**() does not support edge flags.

The mechanics of **glInterleavedArrays**() are intricate and require reference to Example 2-12 and Table 2-5. In that example and table, you'll see et, ec, and en, which are the boolean values for the enabled or disabled texture coordinate, color, and normal arrays, and you'll see st, sc, and sv, which are the sizes (number of components) for the texture coordinate, color, and vertex arrays. tc is the data type for RGBA color, which is the only array that can have non-float interleaved values. pc, pn, and pv are the calculated strides for jumping over individual color, normal, and vertex values, and s is the stride (if one is not specified by the user) to jump from one array element to the next.

The effect of **glInterleavedArrays**() is the same as calling the command sequence in Example 2-12 with many values defined in Table 2-5. All pointer arithmetic is performed in units of **sizeof**(GL_UNSIGNED_BYTE).

Example 2-12 : Effect of glInterleavedArrays(format, stride, pointer)

```
int str;
/* set et, ec, en, st, sc, sv, tc, pc, pn, pv, and s
 * as a function of Table 2-5 and the value of format
 */
str = stride;
if (str == 0)
  str = s;
glDisableClientState(GL_EDGE_FLAG_ARRAY);
glDisableClientState(GL_INDEX_ARRAY);
if (et) {
   glEnableClientState(GL_TEXTURE_COORD_ARRAY);
  glTexCoordPointer(st, GL_FLOAT, str, pointer);
}
else
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);
if (ec) {
  glEnableClientState(GL_COLOR_ARRAY);
```

```
glColorPointer(sc, tc, str, pointer+pc);
}
else
  glDisableClientState(GL_COLOR_ARRAY);
if (en) {
  glEnableClientState(GL_NORMAL_ARRAY);
  glNormalPointer(GL_FLOAT, str, pointer+pn);
}
else
  glDisableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(sv, GL_FLOAT, str, pointer+pv);
```

In Table 2-5, T and F are True and False. f is **sizeof**(GL_FLOAT). c is 4 times **sizeof**(GL_UNSIGNED_BYTE), rounded up to the nearest multiple of f.

format	et	ec	en	st	sc	sv	tc	рс	pn
GL_V2F	F	F	F			2			
GL_V3F	F	F	F			3			
GL_C4UB_V2F	F	Т	F		4	2	GL_UNSIGNED_BYTE	0	
GL_C4UB_V3F	F	Т	F		4	3	GL_UNSIGNED_BYTE	0	
GL_C3F_V3F	F	Т	F		3	3	GL_FLOAT	0	
GL_N3F_V3F	F	F	Т			3			0
GL_C4F_N3F_V3F	F	Т	Т		4	3	GL_FLOAT	0	4f
GL_T2F_V3F	Т	F	F	2		3			
GL_T4F_V4F	Т	F	F	4		4			

Table 2-5 : (continued) Variables that Direct glInterleavedArrays()

GL_T2F_C4UB_V3F	Т	Т	F	2	4	3	GL_UNSIGNED_BYTE	2f	
GL_T2F_C3F_V3F	Т	Т	F	2	3	3	GL_FLOAT	2f	
GL_T2F_N3F_V3F	Т	F	Т	2		3			2f
GL_T2F_C4F_N3F_V3F	Т	Т	Т	2	4	3	GL_FLOAT	2f	6f
GL_T4F_C4F_N3F_V4F	Т	Т	Т	4	4	4	GL_FLOAT	4f	8f

Start by learning the simpler formats, GL_V2F, GL_V3F, and GL_C3F_V3F. If you use any of the formats with C4UB, you may have to use a struct data type or do some delicate type casting and pointer math to pack four unsigned bytes into a single 32-bit word.

For some OpenGL implementations, use of interleaved arrays may increase application performance. With an interleaved array, the exact layout of your data is known. You know your data is tightly packed and may be accessed in one chunk. If interleaved arrays are not used, the stride and size information has to be examined to detect whether data is tightly packed.

Note: glInterleavedArrays() only enables and disables vertex arrays and specifies values for the vertex-array data. It does not render anything. You must still complete Step 3 and call **glArrayElement()**, **glDrawElements()**, or **glDrawArrays()** to dereference the pointers and render graphics.

Attribute Groups

In "Basic State Management," you saw how to set or query an individual state or state variable. Well, you can also save and restore the values of a collection of related state variables with a single command.

OpenGL groups related state variables into an attribute group. For example, the GL_LINE_BIT attribute consists of five state variables: the line width, the GL_LINE_STIPPLE enable status, the line stipple pattern, the line stipple repeat counter, and the GL_LINE_SMOOTH enable status. (See "Antialiasing" in Chapter 6.) With the commands **glPushAttrib**() and **glPopAttrib**(), you can save and restore all five state variables, all at once.

Some state variables are in more than one attribute group. For example, the state variable, GL_CULL_FACE, is part of both the polygon and the enable attribute groups.

In OpenGL Version 1.1, there are now two different attribute stacks. In addition to the original attribute stack (which saves the values of server state variables), there is also a client attribute stack, accessible by

the commands glPushClientAttrib() and glPopClientAttrib().

In general, it's faster to use these commands than to get, save, and restore the values yourself. Some values might be maintained in the hardware, and getting them might be expensive. Also, if you're operating on a remote client, all the attribute data has to be transferred across the network connection and back as it is obtained, saved, and restored. However, your OpenGL implementation keeps the attribute stack on the server, avoiding unnecessary network delays.

There are about twenty different attribute groups, which can be saved and restored by **glPushAttrib**() and **glPopAttrib**(). There are two client attribute groups, which can be saved and restored by **glPushClientAttrib**() and **glPopClientAttrib**(). For both server and client, the attributes are stored on a stack, which has a depth of at least 16 saved attribute groups. (The actual stack depths for your implementation can be obtained using GL_MAX_ATTRIB_STACK_DEPTH and GL_MAX_CLIENT_ATTRIB_STACK_DEPTH with **glGetIntegerv**().) Pushing a full stack or popping an empty one generates an error.

(See the tables in Appendix B to find out exactly which attributes are saved for particular mask values; that is, which attributes are in a particular attribute group.)

void glPushAttrib(GLbitfield mask);

void glPopAttrib(void);

glPushAttrib() saves all the attributes indicated by bits in mask by pushing them onto the attribute stack. glPopAttrib() restores the values of those state variables that were saved with the last glPushAttrib(). Table 2-7 lists the possible mask bits that can be logically ORed together to save any combination of attributes. Each bit corresponds to a collection of individual state variables. For example, GL_LIGHTING_BIT refers to all the state variables related to lighting, which include the current material color, the ambient, diffuse, specular, and emitted light, a list of the lights that are enabled, and the directions of the spotlights. When glPopAttrib() is called, all those variables are restored.

The special mask, GL_ALL_ATTRIB_BITS, is used to save and restore all the state variables in all the attribute groups.

Mask Bit	Attribute Group
GL_ACCUM_BUFFER_BIT	accum-buffer
GL_ALL_ATTRIB_BITS	
GL_COLOR_BUFFER_BIT	color-buffer
GL_CURRENT_BIT	current
GL_CURRENT_BIT	current

depth-buffer
enable
eval
fog
hint
lighting
line
list
pixel
point
polygon
polygon-stipple
scissor
stencil-buffer
texture
transform
viewport

void glPushClientAttrib(GLbitfield mask);

void glPopClientAttrib(void);

glPushClientAttrib() saves all the attributes indicated by bits in mask by pushing them onto the client attribute stack. glPopClientAttrib() restores the values of those state variables that were saved with the last glPushClientAttrib(). Table 2-7 lists the possible mask bits that can be logically ORed together to save any combination of client attributes.

There are two client attribute groups, feedback and select, that cannot be saved or restored with the stack mechanism.

Table 2-7 : Client Attribute Groups

Mask Bit	Attribute Group
GL_CLIENT_PIXEL_STORE_BIT	pixel-store
GL_CLIENT_VERTEX_ARRAY_BIT	vertex-array
GL_ALL_CLIENT_ATTRIB_BITS	
can't be pushed or popped	feedback
can't be pushed or popped	select

Some Hints for Building Polygonal Models of Surfaces

Following are some techniques that you might want to use as you build polygonal approximations of surfaces. You might want to review this section after you've read Chapter 5 on lighting and Chapter 7 on display lists. The lighting conditions affect how models look once they're drawn, and some of the following techniques are much more efficient when used in conjunction with display lists. As you read these techniques, keep in mind that when lighting calculations are enabled, normal vectors must be specified to get proper results.

Constructing polygonal approximations to surfaces is an art, and there is no substitute for experience. This section, however, lists a few pointers that might make it a bit easier to get started.

- Keep polygon orientations consistent. Make sure that when viewed from the outside, all the polygons on the surface are oriented in the same direction (all clockwise or all counterclockwise). Consistent orientation is important for polygon culling and two-sided lighting. Try to get this right the first time, since it's excruciatingly painful to fix the problem later. (If you use **glScale***() to reflect geometry around some axis of symmetry, you might change the orientation with **glFrontFace**() to keep the orientations consistent.)
- When you subdivide a surface, watch out for any nontriangular polygons. The three vertices of a triangle are guaranteed to lie on a plane; any polygon with four or more vertices might not. Nonplanar polygons can be viewed from some orientation such that the edges cross each other, and OpenGL might not render such polygons correctly.
- There's always a trade-off between the display speed and the quality of the image. If you subdivide a surface into a small number of polygons, it renders quickly but might have a jagged appearance; if you subdivide it into millions of tiny polygons, it probably looks good but might

take a long time to render. Ideally, you can provide a parameter to the subdivision routines that indicates how fine a subdivision you want, and if the object is farther from the eye, you can use a coarser subdivision. Also, when you subdivide, use large polygons where the surface is relatively flat, and small polygons in regions of high curvature.

- For high-quality images, it's a good idea to subdivide more on the silhouette edges than in the interior. If the surface is to be rotated relative to the eye, this is tougher to do, since the silhouette edges keep moving. Silhouette edges occur where the normal vectors are perpendicular to the vector from the surface to the viewpoint that is, when their vector dot product is zero. Your subdivision algorithm might choose to subdivide more if this dot product is near zero.
- Try to avoid T-intersections in your models (see Figure 2-16). As shown, there's no guarantee that the line segments AB and BC lie on exactly the same pixels as the segment AC. Sometimes they do, and sometimes they don't, depending on the transformations and orientation. This can cause cracks to appear intermittently in the surface.

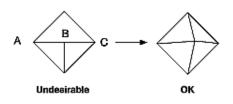


Figure 2-16 : Modifying an Undesirable T-intersection

• If you're constructing a closed surface, make sure to use exactly the same numbers for coordinates at the beginning and end of a closed loop, or you can get gaps and cracks due to numerical round-off. Here's a two-dimensional example of bad code:

```
/* don't use this code */
#define PI 3.14159265
#define EDGES 30
/* draw a circle */
glBegin(GL_LINE_STRIP);
for (i = 0; i <= EDGES; i++)
    glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
glEnd();</pre>
```

The edges meet exactly only if your machine manages to calculate the sine and cosine of 0 and of (2*PI*EDGES/EDGES) and gets exactly the same values. If you trust the floating-point unit on your machine to do this right, the authors have a bridge they'd like to sell you.... To correct the code, make sure that when i == EDGES, you use 0 for the sine and cosine, not 2*PI*EDGES/EDGES. (Or simpler still, use GL_LINE_LOOP instead of GL_LINE_STRIP, and change the loop termination condition to i < EDGES.)

An Example: Building an Icosahedron

To illustrate some of the considerations that arise in approximating a surface, let's look at some example code sequences. This code concerns the vertices of a regular icosahedron (which is a Platonic solid composed of twenty faces that span twelve vertices, each face of which is an equilateral triangle). An icosahedron can be considered a rough approximation for a sphere. Example 2-13 defines the vertices and triangles making up an icosahedron and then draws the icosahedron.

Example 2-13 : Drawing an Icosahedron

```
#define X .525731112119133606
#define Z .850650808352039932
static GLfloat vdata[12][3] =
    };
static GLuint tindices[20][3] = {
     \{0,4,1\}, \{0,9,4\}, \{9,5,4\}, \{4,5,8\}, \{4,8,1\}, \\ \{8,10,1\}, \{8,3,10\}, \{5,3,8\}, \{5,2,3\}, \{2,7,3\}, \\ \{7,10,3\}, \{7,6,10\}, \{7,11,6\}, \{11,0,6\}, \{0,1,6\}, \\ \{6,1,10\}, \{9,0,11\}, \{9,11,2\}, \{9,2,5\}, \{7,2,11\} \}; 
int i;
glBegin(GL TRIANGLES);
for (i = 0; i < 20; i++) {
    /* color information here */
   glVertex3fv(&vdata[tindices[i][0]][0]);
   glVertex3fv(&vdata[tindices[i][1]][0]);
   glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();
```

The strange numbers X and Z are chosen so that the distance from the origin to any of the vertices of the icosahedron is 1.0. The coordinates of the twelve vertices are given in the array vdata[][], where the zeroth vertex is {- &Xgr;, 0.0, &Zgr;}, the first is {X, 0.0, Z}, and so on. The array tindices[][] tells how to link the vertices to make triangles. For example, the first triangle is made from the zeroth, fourth, and first vertex. If you take the vertices for triangles in the order given, all the triangles have the same orientation.

The line that mentions color information should be replaced by a command that sets the color of the *i*th face. If no code appears here, all faces are drawn in the same color, and it'll be impossible to discern the three-dimensional quality of the object. An alternative to explicitly specifying colors is to define surface normals and use lighting, as described in the next subsection.

Note: In all the examples described in this section, unless the surface is to be drawn only once, you should probably save the calculated vertex and normal coordinates so that the calculations don't need to be repeated each time that the surface is drawn. This can be done using your own data structures or by constructing display lists. (See Chapter 7.)

Calculating Normal Vectors for a Surface

If a surface is to be lit, you need to supply the vector normal to the surface. Calculating the normalized cross product of two vectors on that surface provides normal vector. With the flat surfaces of an

icosahedron, all three vertices defining a surface have the same normal vector. In this case, the normal needs to be specified only once for each set of three vertices. The code in Example 2-14 can replace the "color information here" line in Example 2-13 for drawing the icosahedron.

Example 2-14 : Generating Normal Vectors for a Surface

```
GLfloat d1[3], d2[3], norm[3];
for (j = 0; j < 3; j++) {
    d1[j] = vdata[tindices[i][0]][j] - vdata[tindices[i][1]][j];
    d2[j] = vdata[tindices[i][1]][j] - vdata[tindices[i][2]][j];
}
normcrossprod(d1, d2, norm);
glNormal3fv(norm);
```

The function **normcrossprod**() produces the normalized cross product of two vectors, as shown in Example 2-15.

Example 2-15 : Calculating the Normalized Cross Product of Two Vectors

```
void normalize(float v[3]) {
   GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
   if (d == 0.0) {
      error("zero length vector");
     return;
   v[0] /= d; v[1] /= d; v[2] /= d;
}
void normcrossprod(float v1[3], float v2[3], float out[3])
ł
   GLint i, j;
   GLfloat length;
   out[0] = v1[1]*v2[2] - v1[2]*v2[1];
   out[1] = v1[2]*v2[0] - v1[0]*v2[2];
   out[2] = v1[0]*v2[1] - v1[1]*v2[0];
  normalize(out);
}
```

If you're using an icosahedron as an approximation for a shaded sphere, you'll want to use normal vectors that are perpendicular to the true surface of the sphere, rather than being perpendicular to the faces. For a sphere, the normal vectors are simple; each points in the same direction as the vector from the origin to the corresponding vertex. Since the icosahedron vertex data is for an icosahedron of radius 1, the normal and vertex data is identical. Here is the code that would draw an icosahedral approximation of a smoothly shaded sphere (assuming that lighting is enabled, as described in Chapter 5):

```
glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
    glNormal3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i][0]][0]);
    glNormal3fv(&vdata[tindices[i][1]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
    glNormal3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
```

glEnd();

Improving the Model

A twenty-sided approximation to a sphere doesn't look good unless the image of the sphere on the screen is quite small, but there's an easy way to increase the accuracy of the approximation. Imagine the icosahedron inscribed in a sphere, and subdivide the triangles as shown in Figure 2-17. The newly introduced vertices lie slightly inside the sphere, so push them to the surface by normalizing them (dividing them by a factor to make them have length 1). This subdivision process can be repeated for arbitrary accuracy. The three objects shown in Figure 2-17 use 20, 80, and 320 approximating triangles, respectively.

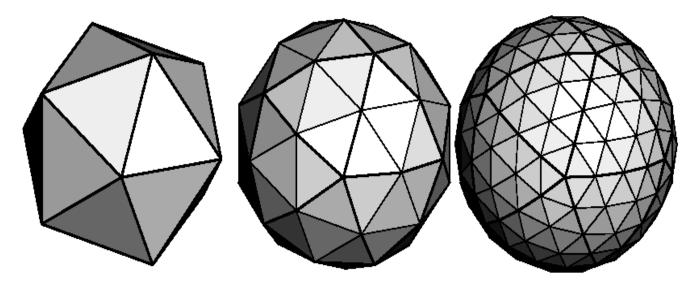


Figure 2-17 : Subdividing to Improve a Polygonal Approximation to a Surface

Example 2-16 performs a single subdivision, creating an 80-sided spherical approximation.

Example 2-16 : Single Subdivision

```
void drawtriangle(float *v1, float *v2, float *v3)
{
    glBegin(GL_TRIANGLES);
        glNormal3fv(v1); vlVertex3fv(v1);
        glNormal3fv(v2); vlVertex3fv(v2);
        glNormal3fv(v3); vlVertex3fv(v3);
    glEnd();
}
void subdivide(float *v1, float *v2, float *v3)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;
    for (i = 0; i < 3; i++) {
        v12[i] = v1[i]+v2[i];
        v23[i] = v2[i]+v3[i];
    }
</pre>
```

Example 2-17 is a slight modification of Example 2-16 which recursively subdivides the triangles to the proper depth. If the depth value is 0, no subdivisions are performed, and the triangle is drawn as is. If the depth is 1, a single subdivision is performed, and so on.

Example 2-17 : Recursive Subdivision

```
void subdivide(float *v1, float *v2, float *v3, long depth)
   GLfloat v12[3], v23[3], v31[3];
   GLint i;
   if (depth == 0) {
      drawtriangle(v1, v2, v3);
      return;
   for (i = 0; i < 3; i++) {
      v12[i] = v1[i]+v2[i];
      v23[i] = v2[i]+v3[i];
      v31[i] = v3[i]+v1[i];
   }
  normalize(v12);
   normalize(v23);
   normalize(v31);
   subdivide(v1, v12, v31, depth-1);
   subdivide(v2, v23, v12, depth-1);
   subdivide(v3, v31, v23, depth-1);
   subdivide(v12, v23, v31, depth-1);
}
```

Generalized Subdivision

A recursive subdivision technique such as the one described in Example 2-17 can be used for other types of surfaces. Typically, the recursion ends either if a certain depth is reached or if some condition on the curvature is satisfied (highly curved parts of surfaces look better with more subdivision).

To look at a more general solution to the problem of subdivision, consider an arbitrary surface parameterized by two variables u[0] and u[1]. Suppose that two routines are provided:

```
void surf(GLfloat u[2], GLfloat vertex[3], GLfloat normal[3]);
float curv(GLfloat u[2]);
```

If **surf**() is passed u[], the corresponding three-dimensional vertex and normal vectors (of length 1) are returned. If u[] is passed to **curv**(), the curvature of the surface at that point is calculated and returned. (See an introductory textbook on differential geometry for more information about measuring surface curvature.)

Example 2-18 shows the recursive routine that subdivides a triangle either until the maximum depth is reached or until the maximum curvature at the three vertices is less than some cutoff.

Example 2-18 : Generalized Subdivision

```
void subdivide(float u1[2], float u2[2], float u3[2],
                float cutoff, long depth)
{
   GLfloat v1[3], v2[3], v3[3], n1[3], n2[3], n3[3];
   GLfloat u12[2], u23[2], u32[2];
   GLint i;
   if (depth == maxdepth || (curv(u1) < cutoff &&
       curv(u2) < cutoff && curv(u3) < cutoff)) {</pre>
      surf(u1, v1, n1); surf(u2, v2, n2); surf(u3, v3, n3);
      glBegin(GL_POLYGON);
         glNormal3fv(n1); glVertex3fv(v1);
         glNormal3fv(n2); glVertex3fv(v2);
         glNormal3fv(n3); glVertex3fv(v3);
      glEnd();
      return;
   ļ
   for (i = 0; i < 2; i++) {
      u12[i] = (u1[i] + u2[i])/2.0;
      u23[i] = (u2[i] + u3[i])/2.0;
      u31[i] = (u3[i] + u1[i])/2.0;
   }
   subdivide(u1, u12, u31, cutoff, depth+1);
   subdivide(u2, u23, u12, cutoff, depth+1);
   subdivide(u3, u31, u23, cutoff, depth+1);
   subdivide(u12, u23, u31, cutoff, depth+1);
}
```

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 3 Viewing

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- View a *geometric model* in any orientation by transforming it in three-dimensional space
- Control the location in three-dimensional space from which the model is viewed
- Clip undesired portions of the model out of the scene that's to be viewed
- Manipulate the appropriate matrix stacks that control model transformation for viewing and project the model onto the screen
- Combine multiple transformations to mimic sophisticated systems in motion, such as a solar system or an articulated robot arm
- Reverse or mimic the operations of the geometric processing pipeline

Chapter 2 explained how to instruct OpenGL to draw the geometric models you want displayed in your scene. Now you must decide how you want to position the models in the scene, and you must choose a vantage point from which to view the scene. You can use the default positioning and vantage point, but most likely you want to specify them.

Look at the image on the cover of this book. The program that produced that image contained a single geometric description of a building block. Each block was carefully positioned in the scene: Some blocks were scattered on the floor, some were stacked on top of each other on the table, and some were assembled to make the globe. Also, a particular viewpoint had to be chosen. Obviously, we wanted to look at the corner of the room containing the globe. But how far away from the scene - and where exactly - should the viewer be? We wanted to make sure that the final image of the scene contained a good view out the window, that a portion of the floor was visible, and that all the objects in the scene were not only visible but presented in an interesting arrangement. This chapter explains how to use OpenGL to accomplish these tasks: how to position and orient models in three-dimensional space and how to establish the location - also in three-dimensional space - of the viewpoint. All of these factors help determine exactly what image appears on the screen.

You want to remember that the point of computer graphics is to create a two-dimensional image of three-dimensional objects (it has to be two-dimensional because it's drawn on a flat screen), but you need to think in three-dimensional coordinates while making many of the decisions that determine what

gets drawn on the screen. A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize three-dimensional space. Create your models in some three-dimensional universe that lies deep inside your computer, and let the computer do its job of calculating which pixels to color.

A series of three computer operations convert an object's three-dimensional coordinates to pixel positions on the screen.

- Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection. Generally, you use a combination of several transformations to draw a scene.
- Since the scene is rendered on a rectangular window, objects (or parts of objects) that lie outside the window must be clipped. In three-dimensional computer graphics, clipping occurs by throwing out objects on one side of a clipping plane.
- Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as a *viewport* transformation.

This chapter describes all of these operations, and how to control them, in the following major sections:

- "Overview: The Camera Analogy" gives an overview of the transformation process by describing the analogy of taking a photograph with a camera, presents a simple example program that transforms an object, and briefly describes the basic OpenGL transformation commands.
- "Viewing and Modeling Transformations" explains in detail how to specify and to imagine the effect of viewing and modeling transformations. These transformations orient the model and the camera relative to each other to obtain the desired final image.
- "Projection Transformations" describes how to specify the shape and orientation of the *viewing volume*. The viewing volume determines how a scene is projected onto the screen (with a perspective or orthographic projection) and which objects or parts of objects are clipped out of the scene.
- "Viewport Transformation" explains how to control the conversion of three-dimensional model coordinates to screen coordinates.
- "Troubleshooting Transformations" presents some tips for discovering why you might not be getting the desired effect from your modeling, viewing, projection, and viewport transformations.
- "Manipulating the Matrix Stacks" discusses how to save and restore certain transformations. This is particularly useful when you're drawing complicated objects that are built up from simpler ones.
- "Additional Clipping Planes" describes how to specify additional clipping planes beyond those defined by the viewing volume.

- "Examples of Composing Several Transformations" walks you through a couple of more complicated uses for transformations.
- "Reversing or Mimicking Transformations" shows you how to take a transformed point in window coordinates and reverse the transformation to obtain its original object coordinates. The transformation itself (without reversal) can also be emulated.

Overview: The Camera Analogy

The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera. As shown in Figure 3-1, the steps with a camera (or a computer) might be the following.

- 1. Set up your tripod and pointing the camera at the scene (viewing transformation).
- 2. Arrange the scene to be photographed into the desired composition (modeling transformation).
- 3. Choose a camera lens or adjust the zoom (projection transformation).
- 4. Determine how large you want the final photograph to be for example, you might want it enlarged (viewport transformation).

After these steps are performed, the picture can be snapped or the scene can be drawn.

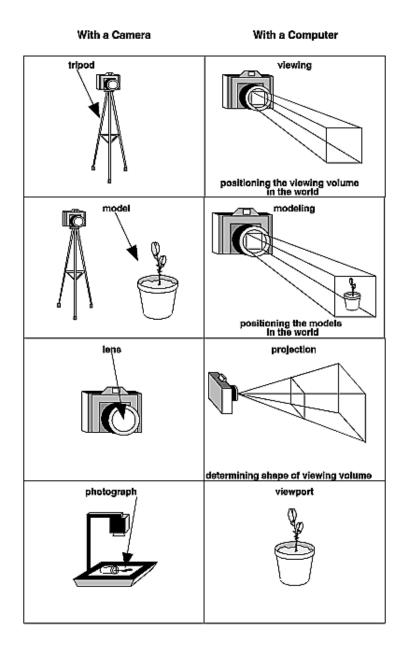


Figure 3-1 : The Camera Analogy

Note that these steps correspond to the order in which you specify the desired transformations in your program, not necessarily the order in which the relevant mathematical operations are performed on an object's vertices. The viewing transformations must precede the modeling transformations in your code, but you can specify the projection and viewport transformations at any point before drawing occurs. Figure 3-2 shows the order in which these operations occur on your computer.

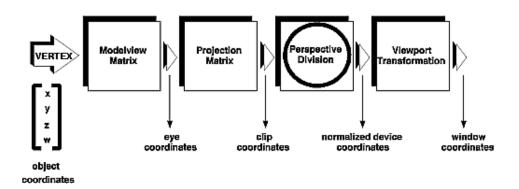


Figure 3-2 : Stages of Vertex Transformation

To specify viewing, modeling, and projection transformations, you construct a 4×4 matrix **M**, which is then multiplied by the coordinates of each vertex *v* in the scene to accomplish the transformation

v'=Mv

(Remember that vertices always have four coordinates (x, y, z, w), though in most cases w is 1 and for two-dimensional data z is 0.) Note that viewing and modeling transformations are automatically applied to surface normal vectors, in addition to vertices. (Normal vectors are used only in eye coordinates.) This ensures that the normal vector's relationship to the vertex data is properly preserved.

The viewing and modeling transformations you specify are combined to form the modelview matrix, which is applied to the incoming object coordinates to yield eye coordinates. Next, if you've specified additional clipping planes to remove certain objects from the scene or to provide cutaway views of objects, these clipping planes are applied.

After that, OpenGL applies the projection matrix to yield *clip coordinates*. This transformation defines a viewing volume; objects outside this volume are clipped so that they're not drawn in the final scene. After this point, the perspective division is performed by dividing coordinate values by *w*, to produce *normalized device coordinates*. (See Appendix F for more information about the meaning of the *w* coordinate and how it affects matrix transformations.) Finally, the transformed coordinates are converted to window coordinates by applying the viewport transformation. You can manipulate the dimensions of the viewport to cause the final image to be enlarged, shrunk, or stretched.

You might correctly suppose that the *x* and *y* coordinates are sufficient to determine which pixels need to be drawn on the screen. However, all the transformations are performed on the *z* coordinates as well. This way, at the end of this transformation process, the *z* values correctly reflect the depth of a given vertex (measured in distance away from the screen). One use for this depth value is to eliminate unnecessary drawing. For example, suppose two vertices have the same *x* and *y* values but different *z* values. OpenGL can use this information to determine which surfaces are obscured by other surfaces and can then avoid drawing the hidden surfaces. (See Chapter 10 for more information about this technique, which is called *hidden-surface removal*.)

As you've probably guessed by now, you need to know a few things about matrix mathematics to get the most out of this chapter. If you want to brush up on your knowledge in this area, you might consult a

textbook on linear algebra.

A Simple Example: Drawing a Cube

Example 3-1 draws a cube that's scaled by a modeling transformation (see Figure 3-3). The viewing transformation, **gluLookAt**(), positions and aims the camera towards where the cube is drawn. A projection transformation and a viewport transformation are also specified. The rest of this section walks you through Example 3-1 and briefly explains the transformation commands it uses. The succeeding sections contain the complete, detailed discussion of all OpenGL's transformation commands.



Figure 3-3 : Transformed Cube

```
Example 3-1 : Transformed Cube: cube.c
```

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}
void display(void)
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
                                  /* clear the matrix */
   glLoadIdentity ();
           /* viewing transformation */
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glScalef (1.0, 2.0, 1.0); /* modeling transformation */
   glutWireCube (1.0);
   glFlush ();
}
void reshape (int w, int h)
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
  glMatrixMode (GL_MODELVIEW);
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
```

```
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

The Viewing Transformation

Recall that the viewing transformation is analogous to positioning and aiming a camera. In this code example, before the viewing transformation can be specified, the current matrix is set to the identity matrix with **glLoadIdentity**(). This step is necessary since most of the transformation commands multiply the current matrix by the specified matrix and then set the result to be the current matrix. If you don't clear the current matrix by loading it with the identity matrix, you continue to combine previous transformation matrices with the new one you supply. In some cases, you do want to perform such combinations, but you also need to clear the matrix sometimes.

In Example 3-1, after the matrix is initialized, the viewing transformation is specified with **gluLookAt(**). The arguments for this command indicate where the camera (or eye position) is placed, where it is aimed, and which way is up. The arguments used here place the camera at (0, 0, 5), aim the camera lens towards (0, 0, 0), and specify the up-vector as (0, 1, 0). The up-vector defines a unique orientation for the camera.

If **gluLookAt(**) was not called, the camera has a default position and orientation. By default, the camera is situated at the origin, points down the negative *z*-axis, and has an up-vector of (0, 1, 0). So in Example 3-1, the overall effect is that **gluLookAt(**) moves the camera 5 units along the *z*-axis. (See "Viewing and Modeling Transformations" for more information about viewing transformations.)

The Modeling Transformation

You use the modeling transformation to position and orient the model. For example, you can rotate, translate, or scale the model - or perform some combination of these operations. In Example 3-1, **glScalef()** is the modeling transformation that is used. The arguments for this command specify how scaling should occur along the three axes. If all the arguments are 1.0, this command has no effect. In Example 3-1, the cube is drawn twice as large in the *y* direction. Thus, if one corner of the cube had originally been at (3.0, 3.0, 3.0), that corner would wind up being drawn at (3.0, 6.0, 3.0). The effect of this modeling transformation is to transform the cube so that it isn't a cube but a rectangular box.

Try This

Change the **gluLookAt**() call in Example 3-1 to the modeling transformation **glTranslatef**() with parameters (0.0, 0.0, -5.0). The result should look exactly the same as when you used **gluLookAt**(). Why are the effects of these two commands similar?

Note that instead of moving the camera (with a viewing transformation) so that the cube could be viewed, you could have moved the cube away from the camera (with a modeling transformation). This

duality in the nature of viewing and modeling transformations is why you need to think about the effect of both types of transformations simultaneously. It doesn't make sense to try to separate the effects, but sometimes it's easier to think about them one way rather than the other. This is also why modeling and viewing transformations are combined into the *modelview matrix* before the transformations are applied. (See "Viewing and Modeling Transformations" for more detail on how to think about modeling and viewing transformations and how to specify them to get the result you want.)

Also note that the modeling and viewing transformations are included in the **display**() routine, along with the call that's used to draw the cube, **glutWireCube**(). This way, **display**() can be used repeatedly to draw the contents of the window if, for example, the window is moved or uncovered, and you've ensured that each time, the cube is drawn in the desired way, with the appropriate transformations. The potential repeated use of **display**() underscores the need to load the identity matrix before performing the viewing and modeling transformations, especially when other transformations might be performed between calls to **display**().

The Projection Transformation

Specifying the projection transformation is like choosing a lens for a camera. You can think of this transformation as determining what the field of view or viewing volume is and therefore what objects are inside it and to some extent how they look. This is equivalent to choosing among wide-angle, normal, and telephoto lenses, for example. With a wide-angle lens, you can include a wider scene in the final photograph than with a telephoto lens, but a telephoto lens allows you to photograph objects as though they're closer to you than they actually are. In computer graphics, you don't have to pay \$10,000 for a 2000-millimeter telephoto lens; once you've bought your graphics workstation, all you need to do is use a smaller number for your field of view.

In addition to the field-of-view considerations, the projection transformation determines how objects are *projected* onto the screen, as its name suggests. Two basic types of projections are provided for you by OpenGL, along with several corresponding commands for describing the relevant parameters in different ways. One type is the *perspective* projection, which matches how you see things in daily life. Perspective makes objects that are farther away appear smaller; for example, it makes railroad tracks appear to converge in the distance. If you're trying to make realistic pictures, you'll want to choose perspective projection, which is specified with the **glFrustum**() command in this code example.

The other type of projection is orthographic, which maps objects directly onto the screen without affecting their relative size. Orthographic projection is used in architectural and computer-aided design applications where the final image needs to reflect the measurements of objects rather than how they might look. Architects create perspective drawings to show how particular buildings or interior spaces look when viewed from various vantage points; the need for orthographic projection arises when blueprint plans or elevations are generated, which are used in the construction of buildings. (See "Projection Transformations" for a discussion of ways to specify both kinds of projection transformations.)

Before **glFrustum**() can be called to set the projection transformation, some preparation needs to happen. As shown in the **reshape**() routine in Example 3-1, the command called **glMatrixMode**() is used first, with the argument GL_PROJECTION. This indicates that the current matrix specifies the projection transformation; the following transformation calls then affect the projection matrix. As you can see, a few lines later **glMatrixMode**() is called again, this time with GL_MODELVIEW as the

argument. This indicates that succeeding transformations now affect the modelview matrix instead of the projection matrix. (See "Manipulating the Matrix Stacks" for more information about how to control the projection and modelview matrices.)

Note that **glLoadIdentity**() is used to initialize the current projection matrix so that only the specified projection transformation has an effect. Now **glFrustum**() can be called, with arguments that define the parameters of the projection transformation. In this example, both the projection transformation and the viewport transformation are contained in the **reshape**() routine, which is called when the window is first created and whenever the window is moved or reshaped. This makes sense, since both projecting (the width to height aspect ratio of the projection viewing volume) and applying the viewport relate directly to the screen, and specifically to the size or aspect ratio of the window on the screen.

Try This

Change the **glFrustum**() call in Example 3-1 to the more commonly used Utility Library routine **gluPerspective**() with parameters (60.0, 1.0, 1.5, 20.0). Then experiment with different values, especially for *fovy* and *aspect*.

The Viewport Transformation

Together, the projection transformation and the viewport transformation determine how a scene gets mapped onto the computer screen. The projection transformation specifies the mechanics of how the mapping should occur, and the viewport indicates the shape of the available screen area into which the scene is mapped. Since the viewport specifies the region the image occupies on the computer screen, you can think of the viewport transformation as defining the size and location of the final processed photograph - for example, whether the photograph should be enlarged or shrunk.

The arguments to **glViewport**() describe the origin of the available screen space within the window - (0, 0) in this example - and the width and height of the available screen area, all measured in pixels on the screen. This is why this command needs to be called within **reshape**() - if the window changes size, the viewport needs to change accordingly. Note that the width and height are specified using the actual width and height of the window; often, you want to specify the viewport this way rather than giving an absolute size. (See "Viewport Transformation" for more information about how to define the viewport.)

Drawing the Scene

Once all the necessary transformations have been specified, you can draw the scene (that is, take the photograph). As the scene is drawn, OpenGL transforms each vertex of every object in the scene by the modeling and viewing transformations. Each vertex is then transformed as specified by the projection transformation and clipped if it lies outside the viewing volume described by the projection transformation. Finally, the remaining transformed vertices are divided by *w* and mapped onto the viewport.

General-Purpose Transformation Commands

This section discusses some OpenGL commands that you might find useful as you specify desired transformations. You've already seen a couple of these commands, **glMatrixMode**() and **glLoadIdentity**(). The other two commands described here - **glLoadMatrix***() and **glMultMatrix***() -

allow you to specify any transformation matrix directly and then to multiply the current matrix by that specified matrix. More specific transformation commands - such as **gluLookAt()** and **glScale*()** - are described in later sections.

As described in the preceding section, you need to state whether you want to modify the modelview or projection matrix before supplying a transformation command. You choose the matrix with **glMatrixMode()**. When you use nested sets of OpenGL commands that might be called repeatedly, remember to reset the matrix mode correctly. (The **glMatrixMode(**) command can also be used to indicate the texture matrix; texturing is discussed in detail in "The Texture Matrix Stack" in Chapter 9.)

void glMatrixMode(GLenum mode);

Specifies whether the modelview, projection, or texture matrix will be modified, using the argument GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE for mode. Subsequent transformation commands affect the specified matrix. Note that only one matrix can be modified at a time. By default, the modelview matrix is the one that's modifiable, and all three matrices contain the identity matrix.

You use the **glLoadIdentity()** command to clear the currently modifiable matrix for future transformation commands, since these commands modify the current matrix. Typically, you always call this command before specifying projection or viewing transformations, but you might also call it before specifying a modeling transformation.

void glLoadIdentity(void);

Sets the currently modifiable matrix to the 4×4 identity matrix.

If you want to specify explicitly a particular matrix to be loaded as the current matrix, use **glLoadMatrix***(). Similarly, use **glMultMatrix***() to multiply the current matrix by the matrix passed in as an argument. The argument for both these commands is a vector of sixteen values (m1, m2, ..., m16) that specifies a matrix **M** as follows:

 $\mathbf{M} = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$

Remember that you might be able to maximize efficiency by using display lists to store frequently used matrices (and their inverses) rather than recomputing them. (See "Display-List Design Philosophy" in Chapter 7.) (OpenGL implementations often must compute the inverse of the modelview matrix so that normals and clipping planes can be correctly transformed to eye coordinates.)

Caution: If you're programming in C and you declare a matrix as m[4][4], then the element m[i][j] is in the *i*th column and *j*th row of the OpenGL transformation matrix. This is the reverse of the standard C convention in which m[i][j] is in row *i* and column *j*. To avoid confusion, you should declare your matrices as m[16].

void glLoadMatrix{fd}(const TYPE *m);

Sets the sixteen values of the current matrix to those specified by m.

void glMultMatrix{fd}(const TYPE *m);

Multiplies the matrix specified by the sixteen values pointed to by m by the current matrix and stores the result as the current matrix.

Note: All matrix multiplication with OpenGL occurs as follows: Suppose the current matrix is **C** and the matrix specified with **glMultMatrix***() or any of the transformation commands is **M**. After multiplication, the final matrix is always **CM**. Since matrix multiplication isn't generally commutative, the order makes a difference.

Viewing and Modeling Transformations

Viewing and modeling transformations are inextricably related in OpenGL and are in fact combined into a single modelview matrix. (See "A Simple Example: Drawing a Cube.") One of the toughest problems newcomers to computer graphics face is understanding the effects of combined three-dimensional transformations. As you've already seen, there are alternative ways to think about transformations - do you want to move the camera in one direction, or move the object in the opposite direction? Each way of thinking about transformations has advantages and disadvantages, but in some cases one way more naturally matches the effect of the intended transformation. If you can find a natural approach for your particular application, it's easier to visualize the necessary transformations and then write the corresponding code to specify the matrix manipulations. The first part of this section discusses how to think about transformations; later, specific commands are presented. For now, we use only the matrix-manipulation commands you've already seen. Finally, keep in mind that you must call **glMatrixMode**() with GL_MODELVIEW as its argument prior to performing modeling or viewing transformations.

Thinking about Transformations

Let's start with a simple case of two transformations: a 45-degree counterclockwise rotation about the origin around the *z*-axis, and a translation down the *x*-axis. Suppose that the object you're drawing is small compared to the translation (so that you can see the effect of the translation), and that it's originally located at the origin. If you rotate the object first and then translate it, the rotated object appears on the *x*-axis. If you translate it down the *x*-axis first, however, and then rotate about the origin, the object is on the line y=x, as shown in Figure 3-4. In general, the order of transformations is critical. If you do transformation A and then transformation B, you almost always get something different than if you do them in the opposite order.

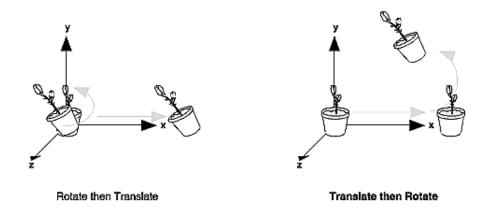


Figure 3-4 : Rotating First or Translating First

Now let's talk about the order in which you specify a series of transformations. All viewing and modeling transformations are represented as 4×4 matrices. Each successive **glMultMatrix***() or transformation command multiplies a new 4×4 matrix **M** by the current modelview matrix **C** to yield **CM**. Finally, vertices *v* are multiplied by the current modelview matrix. This process means that the last transformation command called in your program is actually the first one applied to the vertices: **CMv**. Thus, one way of looking at it is to say that you have to specify the matrices in the reverse order. Like many other things, however, once you've gotten used to thinking about this correctly, backward will seem like forward.

Consider the following code sequence, which draws a single point using three transformations:

With this code, the modelview matrix successively contains I, N, NM, and finally NML, where I represents the identity matrix. The transformed vertex is NMLv. Thus, the vertex transformation is N(M(Lv)) - that is, v is multiplied first by L, the resulting Lv is multiplied by M, and the resulting MLv is multiplied by N. Notice that the transformations to vertex v effectively occur in the opposite order than they were specified. (Actually, only a single multiplication of a vertex by the modelview matrix occurs; in this example, the N, M, and L matrices are already multiplied into a single matrix before it's applied to v.)

Grand, Fixed Coordinate System

Thus, if you like to think in terms of a grand, fixed coordinate system - in which matrix multiplications affect the position, orientation, and scaling of your model - you have to think of the multiplications as occurring in the opposite order from how they appear in the code. Using the simple example shown on the left side of Figure 3-4 (a rotation about the origin and a translation along the *x*-axis), if you want the

object to appear on the axis after the operations, the rotation must occur first, followed by the translation. To do this, you'll need to reverse the order of operations, so the code looks something like this (where \mathbf{R} is the rotation matrix and \mathbf{T} is the translation matrix):

Moving a Local Coordinate System

Another way to view matrix multiplications is to forget about a grand, fixed coordinate system in which your model is transformed and instead imagine that a local coordinate system is tied to the object you're drawing. All operations occur relative to this changing coordinate system. With this approach, the matrix multiplications now appear in the natural order in the code. (Regardless of which analogy you're using, the code is the same, but how you think about it differs.) To see this in the translation-rotation example, begin by visualizing the object with a coordinate system tied to it. The translation operation moves the object and its coordinate system down the *x*-axis. Then, the rotation occurs about the (now-translated) origin, so the object rotates in place in its position on the axis.

This approach is what you should use for applications such as articulated robot arms, where there are joints at the shoulder, elbow, and wrist, and on each of the fingers. To figure out where the tips of the fingers go relative to the body, you'd like to start at the shoulder, go down to the wrist, and so on, applying the appropriate rotations and translations at each joint. Thinking about it in reverse would be far more confusing.

This second approach can be problematic, however, in cases where scaling occurs, and especially so when the scaling is nonuniform (scaling different amounts along the different axes). After uniform scaling, translations move a vertex by a multiple of what they did before, since the coordinate system is stretched. Nonuniform scaling mixed with rotations may make the axes of the local coordinate system nonperpendicular.

As mentioned earlier, you normally issue viewing transformation commands in your program before any modeling transformations. This way, a vertex in a model is first transformed into the desired orientation and then transformed by the viewing operation. Since the matrix multiplications must be specified in reverse order, the viewing commands need to come first. Note, however, that you don't need to specify either viewing or modeling transformations if you're satisfied with the default conditions. If there's no viewing transformation, the "camera" is left in the default position at the origin, pointed toward the negative *z*-axis; if there's no modeling transformation, the model isn't moved, and it retains its specified position, orientation, and size.

Since the commands for performing modeling transformations can be used to perform viewing transformations, modeling transformations are *discussed* first, even if viewing transformations are actually *issued* first. This order for discussion also matches the way many programmers think when planning their code: Often, they write all the code necessary to compose the scene, which involves transformations to position and orient objects correctly relative to each other. Next, they decide where they want the viewpoint to be relative to the scene they've composed, and then they write the viewing transformations accordingly.

Modeling Transformations

The three OpenGL routines for modeling transformations are **glTranslate***(), **glRotate***(), and **glScale***(). As you might suspect, these routines transform an object (or coordinate system, if you're thinking of it that way) by moving, rotating, stretching, shrinking, or reflecting it. All three commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling **glMultMatrix***() with that matrix as the argument. However, these three routines might be faster than using **glMultMatrix***(). OpenGL automatically computes the matrices for you. (See Appendix F if you're interested in the details.)

In the command summaries that follow, each matrix multiplication is described in terms of what it does to the vertices of a geometric object using the fixed coordinate system approach, and in terms of what it does to the local coordinate system that's attached to an object.

Translate

void glTranslate{fd}(TYPEx, TYPE y, TYPEz);

Multiplies the current matrix by a matrix that moves (translates) an object by the given x, y, and z values (or moves the local coordinate system by the same amounts).

Figure 3-5 shows the effect of **glTranslate***().

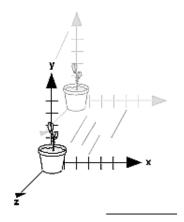


Figure 3-5 : Translating an Object

Note that using (0.0, 0.0, 0.0) as the argument for **glTranslate***() is the identity operation - that is, it has no effect on an object or its local coordinate system.

Rotate

void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);

Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation in degrees.

The effect of **glRotatef**(45.0, 0.0, 0.0, 1.0), which is a rotation of 45 degrees about the *z*-axis, is shown in Figure 3-6.

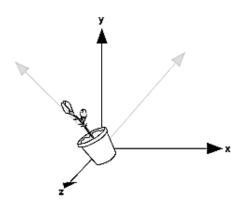


Figure 3-6 : Rotating an Object

Note that an object that lies farther from the axis of rotation is more dramatically rotated (has a larger orbit) than an object drawn near the axis. Also, if the *angle* argument is zero, the **glRotate***() command has no effect.

Scale

```
void glScale{fd}(TYPEx, TYPE y, TYPEz);
```

Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each x, y, and z coordinate of every point in the object is multiplied by the corresponding argument x, y, or z. With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or reflected by the x, y, and z factors, and the associated object is transformed with them.

Figure 3-7 shows the effect of **glScalef**(2.0, -0.5, 1.0).

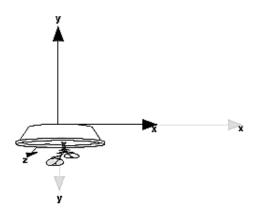


Figure 3-7 : Scaling and Reflecting an Object

glScale*() is the only one of the three modeling transformations that changes the apparent size of an object: Scaling with values greater than 1.0 stretches an object, and using values less than 1.0 shrinks it. Scaling with a -1.0 value reflects an object across an axis. The identity values for scaling are (1.0, 1.0, 1.0). In general, you should limit your use of **glScale***() to those cases where it is necessary. Using **glScale***() decreases the performance of lighting calculations, because the normal vectors have to be renormalized after transformation.

Note: A scale value of zero collapses all object coordinates along that axis to zero. It's usually not a good idea to do this, because such an operation cannot be undone. Mathematically speaking, the matrix cannot be inverted, and inverse matrices are required for certain lighting operations. (See Chapter 5.) Sometimes collapsing coordinates does make sense, however; the calculation of shadows on a planar surface is a typical application. (See "Shadows" in Chapter 14.) In general, if a coordinate system is to be collapsed, the projection matrix should be used rather than the modelview matrix.

A Modeling Transformation Code Example

Example 3-2 is a portion of a program that renders a triangle four times, as shown in Figure 3-8. These are the four transformed triangles.

- A solid wireframe triangle is drawn with no modeling transformation.
- The same triangle is drawn again, but with a dashed line stipple and translated (to the left along the negative x-axis).
- A triangle is drawn with a long dashed line stipple, with its height (y-axis) halved and its width (x-axis) increased by 50%.
- A rotated triangle, made of dotted lines, is drawn.



Figure 3-8 : Modeling Transformation Example

Example 3-2 : Using Modeling Transformations: model.c

```
glLineStipple(1, 0xF00F); /*long dashed lines */
glLoadIdentity();
glScalef(1.5, 0.5, 1.0);
draw_triangle();
glLineStipple(1, 0x8888); /* dotted lines */
glLoadIdentity();
glRotatef (90.0, 0.0, 0.0, 1.0);
draw_triangle ();
glDisable (GL_LINE_STIPPLE);
```

Note the use of **glLoadIdentity**() to isolate the effects of modeling transformations; initializing the matrix values prevents successive transformations from having a cumulative effect. Even though using **glLoadIdentity**() repeatedly has the desired effect, it may be inefficient, because you may have to respecify viewing or modeling transformations. (See "Manipulating the Matrix Stacks" for a better way to isolate transformations.)

Note: Sometimes, programmers who want a continuously rotating object attempt to achieve this by repeatedly applying a rotation matrix that has small values. The problem with this technique is that because of round-off errors, the product of thousands of tiny rotations gradually drifts away from the value you really want (it might even become something that isn't a rotation). Instead of using this technique, increment the angle and issue a new rotation command with the new angle at each update step.

Viewing Transformations

A viewing transformation changes the position and orientation of the viewpoint. If you recall the camera analogy, the viewing transformation positions the camera tripod, pointing the camera toward the model. Just as you move the camera to some position and rotate it until it points in the desired direction, viewing transformations are generally composed of translations and rotations. Also remember that to achieve a certain scene composition in the final image or photograph, you can either move the camera or move all the objects in the opposite direction. Thus, a modeling transformation that rotates an object counterclockwise is equivalent to a viewing transformation that rotates the camera clockwise, for example. Finally, keep in mind that the viewing transformation commands must be called before any modeling transformations are performed, so that the modeling transformations take effect on the objects first.

You can manufacture a viewing transformation in any of several ways, as described next. You can also choose to use the default location and orientation of the viewpoint, which is at the origin, looking down the negative *z*-axis.

- Use one or more modeling transformation commands (that is, **glTranslate***() and **glRotate***()). You can think of the effect of these transformations as moving the camera position or as moving all the objects in the world, relative to a stationary camera.
- Use the Utility Library routine **gluLookAt(**) to define a line of sight. This routine encapsulates a series of rotation and translation commands.
- Create your own utility routine that encapsulates rotations and translations. Some applications might require custom routines that allow you to specify the viewing transformation in a convenient

way. For example, you might want to specify the roll, pitch, and heading rotation angles of a plane in flight, or you might want to specify a transformation in terms of polar coordinates for a camera that's orbiting around an object.

Using glTranslate*() and glRotate*()

When you use modeling transformation commands to emulate viewing transformations, you're trying to move the viewpoint in a desired way while keeping the objects in the world stationary. Since the viewpoint is initially located at the origin and since objects are often most easily constructed there as well (see Figure 3-9), in general you have to perform some transformation so that the objects can be viewed. Note that, as shown in the figure, the camera initially points down the negative *z*-axis. (You're seeing the back of the camera.)

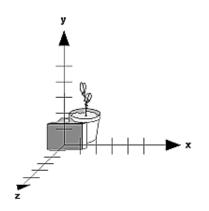


Figure 3-9 : Object and Viewpoint at the Origin

In the simplest case, you can move the viewpoint backward, away from the objects; this has the same effect as moving the objects forward, or away from the viewpoint. Remember that by default forward is down the negative *z*-axis; if you rotate the viewpoint, forward has a different meaning. So, to put 5 units of distance between the viewpoint and the objects by moving the viewpoint, as shown in Figure 3-10, use

```
glTranslatef(0.0, 0.0, -5.0);
```

This routine moves the objects in the scene -5 units along the z axis. This is also equivalent to moving the camera +5 units along the z axis.

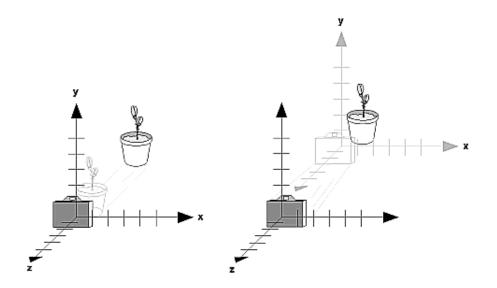


Figure 3-10 : Separating the Viewpoint and the Object

Now suppose you want to view the objects from the side. Should you issue a rotate command before or after the translate command? If you're thinking in terms of a grand, fixed coordinate system, first imagine both the object and the camera at the origin. You could rotate the object first and then move it away from the camera so that the desired side is visible. Since you know that with the fixed coordinate system approach, commands have to be issued in the opposite order in which they should take effect, you know that you need to write the translate command first in your code and follow it with the rotate command.

Now let's use the local coordinate system approach. In this case, think about moving the object and its local coordinate system away from the origin; then, the rotate command is carried out using the now-translated coordinate system. With this approach, commands are issued in the order in which they're applied, so once again the translate command comes first. Thus, the sequence of transformation commands to produce the desired result is

glTranslatef(0.0, 0.0, -5.0); glRotatef(90.0, 0.0, 1.0, 0.0);

If you're having trouble keeping track of the effect of successive matrix multiplications, try using both the fixed and local coordinate system approaches and see whether one makes more sense to you. Note that with the fixed coordinate system, rotations always occur about the grand origin, whereas with the local coordinate system, rotations occur about the origin of the local system. You might also try using the **gluLookAt(**) utility routine described in the next section.

Using the gluLookAt() Utility Routine

Often, programmers construct a scene around the origin or some other convenient location, then they want to look at it from an arbitrary point to get a good view of it. As its name suggests, the **gluLookAt(**) utility routine is designed for just this purpose. It takes three sets of arguments, which specify the location of the viewpoint, define a reference point toward which the camera is aimed, and indicate which

direction is up. Choose the viewpoint to yield the desired view of the scene. The reference point is typically somewhere in the middle of the scene. (If you've built your scene at the origin, the reference point is probably the origin.) It might be a little trickier to specify the correct up-vector. Again, if you've built some real-world scene at or around the origin and if you've been taking the positive *y*-axis to point upward, then that's your up-vector for **gluLookAt(**). However, if you're designing a flight simulator, up is the direction perpendicular to the plane's wings, from the plane toward the sky when the plane is right-side up on the ground.

The **gluLookAt(**) routine is particularly useful when you want to pan across a landscape, for instance. With a viewing volume that's symmetric in both *x* and *y*, the (*eyex, eyey, eyez*) point specified is always in the center of the image on the screen, so you can use a series of commands to move this point slightly, thereby panning across the scene.

void **gluLookAt**(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble upx, GLdouble upy, GLdouble upz);

Defines a viewing matrix and multiplies it to the right of the current matrix. The desired viewpoint is specified by eyex, eyey, and eyez. The centerx, centery, and centerz arguments specify any point along the desired line of sight, but typically they're some point in the center of the scene being looked at. The upx, upy, and upz arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume).

In the default position, the camera is at the origin, is looking down the negative *z*-axis, and has the positive *y*-axis as straight up. This is the same as calling

gluLookat (0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);

The *z* value of the reference point is -100.0, but could be any negative *z*, because the line of sight will remain the same. In this case, you don't actually want to call **gluLookAt(**), because this is the default (see Figure 3-11) and you are already there! (The lines extending from the camera represent the viewing volume, which indicates its field of view.)

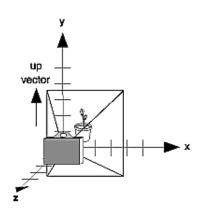


Figure 3-11 : Default Camera Position

Figure 3-12 shows the effect of a typical **gluLookAt**() routine. The camera position (*eyex, eyey, eyez*) is at (4, 2, 1). In this case, the camera is looking right at the model, so the reference point is at (2, 4, -3).

An orientation vector of (2, 2, -1) is chosen to rotate the viewpoint to this 45-degree angle.

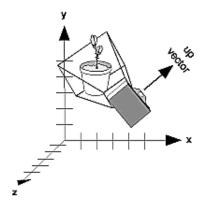


Figure 3-12 : Using gluLookAt()

So, to achieve this effect, call

gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);

Note that **gluLookAt**() is part of the Utility Library rather than the basic OpenGL library. This isn't because it's not useful, but because it encapsulates several basic OpenGL commands - specifically, **glTranslate***() and **glRotate***(). To see this, imagine a camera located at an arbitrary viewpoint and oriented according to a line of sight, both as specified with **gluLookAt**() and a scene located at the origin. To "undo" what **gluLookAt**() does, you need to transform the camera so that it sits at the origin and points down the negative *z*-axis, the default position. A simple translate moves the camera to the origin. You can easily imagine a series of rotations about each of the three axes of a fixed coordinate system that would orient the camera so that it pointed toward negative *z* values. Since OpenGL allows rotation about an arbitrary axis, you can accomplish any desired rotation of the camera with a single **glRotate***() command.

Note: You can have only one active viewing transformation. You cannot try to combine the effects of two viewing transformations, any more than a camera can have two tripods. If you want to change the position of the camera, make sure you call **glLoadIdentity**() to wipe away the effects of any current viewing transformation.

Advanced

To transform any arbitrary vector so that it's coincident with another arbitrary vector (for instance, the negative *z*-axis), you need to do a little mathematics. The axis about which you want to rotate is given by the cross product of the two normalized vectors. To find the angle of rotation, normalize the initial two vectors. The cosine of the desired angle between the vectors is equal to the dot product of the normalized vectors. The axis given by the cross product is always between 0 and 180 degrees. (See Appendix E for definitions of cross and dot products.)

Note that computing the angle between two normalized vectors by taking the inverse cosine of their dot product is not very accurate, especially for small angles. But it should work well enough to get you

started.

Creating a Custom Utility Routine

Advanced

For some specialized applications, you might want to define your own transformation routine. Since this is rarely done and in any case is a fairly advanced topic, it's left mostly as an exercise for the reader. The following exercises suggest two custom viewing transformations that might be useful.

Try This

• Suppose you're writing a flight simulator and you'd like to display the world from the point of view of the pilot of a plane. The world is described in a coordinate system with the origin on the runway and the plane at coordinates (*x*, *y*, *z*). Suppose further that the plane has some *roll*, *pitch*, and *heading* (these are rotation angles of the plane relative to its center of gravity).

Show that the following routine could serve as the viewing transformation:

```
void pilotView{GLdouble planex, GLdouble planey,
GLdouble planez, GLdouble roll,
GLdouble pitch, GLdouble heading)
{
    glRotated(roll, 0.0, 0.0, 1.0);
    glRotated(pitch, 0.0, 1.0, 0.0);
    glRotated(heading, 1.0, 0.0, 0.0);
    glTranslated(-planex, -planey, -planez);
}
```

• Suppose your application involves orbiting the camera around an object that's centered at the origin. In this case, you'd like to specify the viewing transformation by using polar coordinates. Let the *distance* variable define the radius of the orbit, or how far the camera is from the origin. (Initially, the camera is moved *distance* units along the positive *z*-axis.) The *azimuth* describes the angle of rotation of the camera about the object in the *x*-*y* plane, measured from the positive *y*-axis. Similarly, *elevation* is the angle of rotation of the camera in the *y*-*z* plane, measured from the positive *z*-axis. Finally, *twist* represents the rotation of the viewing volume around its line of sight.

Show that the following routine could serve as the viewing transformation:

Projection Transformations

The previous section described how to compose the desired modelview matrix so that the correct modeling and viewing transformations are applied. This section explains how to define the desired projection matrix, which is also used to transform the vertices in your scene. Before you issue any of the transformation commands described in this section, remember to call

glMatrixMode(GL_PROJECTION);
glLoadIdentity();

so that the commands affect the projection matrix rather than the modelview matrix and so that you avoid compound projection transformations. Since each projection transformation command completely describes a particular transformation, typically you don't want to combine a projection transformation with another transformation.

The purpose of the projection transformation is to define a *viewing volume*, which is used in two ways. The viewing volume determines how an object is projected onto the screen (that is, by using a perspective or an orthographic projection), and it defines which objects or portions of objects are clipped out of the final image. You can think of the viewpoint we've been talking about as existing at one end of the viewing volume. At this point, you might want to reread "A Simple Example: Drawing a Cube" for its overview of all the transformations, including projection transformations.

Perspective Projection

The most unmistakable characteristic of perspective projection is foreshortening: the farther an object is from the camera, the smaller it appears in the final image. This occurs because the viewing volume for a perspective projection is a frustum of a pyramid (a truncated pyramid whose top has been cut off by a plane parallel to its base). Objects that fall within the viewing volume are projected toward the apex of the pyramid, where the camera or viewpoint is. Objects that are closer to the viewpoint appear larger because they occupy a proportionally larger amount of the viewing volume than those that are farther away, in the larger part of the frustum. This method of projection is commonly used for animation, visual simulation, and any other applications that strive for some degree of realism because it's similar to how our eye (or a camera) works.

The command to define a frustum, **glFrustum**(), calculates a matrix that accomplishes perspective projection and multiplies the current projection matrix (typically the identity matrix) by it. Recall that the viewing volume is used to clip objects that lie outside of it; the four sides of the frustum, its top, and its base correspond to the six clipping planes of the viewing volume, as shown in Figure 3-13. Objects or parts of objects outside these planes are clipped from the final image. Note that **glFrustum**() doesn't require you to define a symmetric viewing volume.

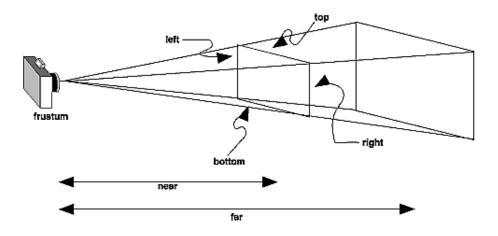


Figure 3-13 : Perspective Viewing Volume Specified by glFrustum()

void **glFrustum**(GLdouble left, GLdouble right, GLdouble bottom,

GLdouble top, GLdouble near, GLdouble far);

Creates a matrix for a perspective-view frustum and multiplies the current matrix by it. The frustum's viewing volume is defined by the parameters: (left, bottom, -near) and (right, top, -near) specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane; near and far give the distances from the viewpoint to the near and far clipping planes. They should always be positive.

The frustum has a default orientation in three-dimensional space. You can perform rotations or translations on the projection matrix to alter this orientation, but this is tricky and nearly always avoidable.

Advanced

Also, the frustum doesn't have to be symmetrical, and its axis isn't necessarily aligned with the *z*-axis. For example, you can use **glFrustum**() to draw a picture as if you were looking through a rectangular window of a house, where the window was above and to the right of you. Photographers use such a viewing volume to create false perspectives. You might use it to have the hardware calculate images at much higher than normal resolutions, perhaps for use on a printer. For example, if you want an image that has twice the resolution of your screen, draw the same picture four times, each time using the frustum to cover the entire screen with one-quarter of the image. After each quarter of the image is rendered, you can read the pixels back to collect the data for the higher-resolution image. (See Chapter 8 for more information about reading pixel data.)

Although it's easy to understand conceptually, **glFrustum**() isn't intuitive to use. Instead, you might try the Utility Library routine **gluPerspective**(). This routine creates a viewing volume of the same shape as **glFrustum**() does, but you specify it in a different way. Rather than specifying corners of the near clipping plane, you specify the angle of the field of view (&THgr; , or theta, in Figure 3-14) in the *y* direction and the aspect ratio of the width to height (x/y). (For a square portion of the screen, the aspect ratio is 1.0.) These two parameters are enough to determine an untruncated pyramid along the line of sight, as shown in Figure 3-14. You also specify the distance between the viewpoint and the near and far

clipping planes, thereby truncating the pyramid. Note that **gluPerspective**() is limited to creating frustums that are symmetric in both the *x*- and *y*-axes along the line of sight, but this is usually what you want.

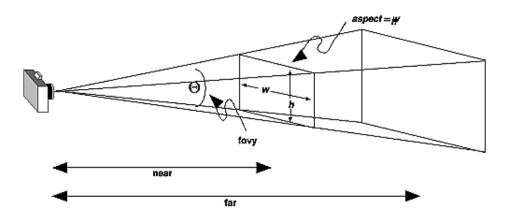


Figure 3-14 : Perspective Viewing Volume Specified by gluPerspective()

void **gluPerspective**(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far);

Creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. fovy is the angle of the field of view in the x-z plane; its value must be in the range [0.0,180.0]. aspect is the aspect ratio of the frustum, its width divided by its height. near and far values the distances between the viewpoint and the clipping planes, along the negative z-axis. They should always be positive.

Just as with **glFrustum**(), you can apply rotations or translations to change the default orientation of the viewing volume created by **gluPerspective**(). With no such transformations, the viewpoint remains at the origin, and the line of sight points down the negative *z*-axis.

With **gluPerspective**(), you need to pick appropriate values for the field of view, or the image may look distorted. For example, suppose you're drawing to the entire screen, which happens to be 11 inches high. If you choose a field of view of 90 degrees, your eye has to be about 7.8 inches from the screen for the image to appear undistorted. (This is the distance that makes the screen subtend 90 degrees.) If your eye is farther from the screen, as it usually is, the perspective doesn't look right. If your drawing area occupies less than the full screen, your eye has to be even closer. To get a perfect field of view, figure out how far your eye normally is from the screen and how big the window is, and calculate the angle the window subtends at that size and distance. It's probably smaller than you would guess. Another way to think about it is that a 94-degree field of view with a 35-millimeter camera requires a 20-millimeter lens, which is a very wide-angle lens. (See "Troubleshooting Transformations" for more details on how to calculate the desired field of view.)

The preceding paragraph mentions inches and millimeters - do these really have anything to do with OpenGL? The answer is, in a word, no. The projection and other transformations are inherently unitless. If you want to think of the near and far clipping planes as located at 1.0 and 20.0 meters, inches, kilometers, or leagues, it's up to you. The only rule is that you have to use a consistent unit of

measurement. Then the resulting image is drawn to scale.

Orthographic Projection

With an orthographic projection, the viewing volume is a rectangular parallelepiped, or more informally, a box (see Figure 3-15). Unlike perspective projection, the size of the viewing volume doesn't change from one end to the other, so distance from the camera doesn't affect how large an object appears. This type of projection is used for applications such as creating architectural blueprints and computer-aided design, where it's crucial to maintain the actual sizes of objects and angles between them as they're projected.

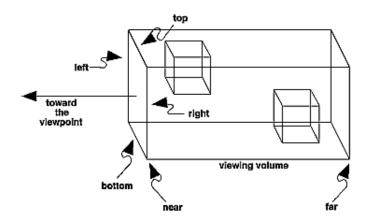


Figure 3-15 : Orthographic Viewing Volume

The command **glOrtho**() creates an orthographic parallel viewing volume. As with **glFrustum**(), you specify the corners of the near clipping plane and the distance to the far clipping plane.

void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,

GLdouble top, GLdouble near, GLdouble far);

Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (left, bottom, -near) and (right, top, -near) are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewport window, respectively. (left, bottom, -far) and (right, top, -far) are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both near and far can be positive or negative.

With no other transformations, the direction of projection is parallel to the *z*-axis, and the viewpoint faces toward the negative *z*-axis. Note that this means that the values passed in for *far* and *near* are used as negative *z* values if these planes are in front of the viewpoint, and positive if they're behind the viewpoint.

For the special case of projecting a two-dimensional image onto a two-dimensional screen, use the Utility Library routine **gluOrtho2D**(). This routine is identical to the three-dimensional version, **glOrtho**(), except that all the *z* coordinates for objects in the scene are assumed to lie between -1.0 and 1.0. If you're drawing two-dimensional objects using the two-dimensional vertex commands, all the *z* coordinates are zero; thus, none of the objects are clipped because of their *z* values.

void **gluOrtho2D**(*GLdouble left, GLdouble right, GLdouble bottom, GLdouble top*);

Creates a matrix for projecting two-dimensional coordinates onto the screen and multiplies the current projection matrix by it. The clipping region is a rectangle with the lower-left corner at (left, bottom) and the upper-right corner at (right, top).

Viewing Volume Clipping

After the vertices of the objects in the scene have been transformed by the modelview and projection matrices, any primitives that lie outside the viewing volume are clipped. The six clipping planes used are those that define the sides and ends of the viewing volume. You can specify additional clipping planes and locate them wherever you choose. (See "Additional Clipping Planes" for information about this relatively advanced topic.) Keep in mind that OpenGL reconstructs the edges of polygons that get clipped.

Viewport Transformation

Recalling the camera analogy, you know that the viewport transformation corresponds to the stage where the size of the developed photograph is chosen. Do you want a wallet-size or a poster-size photograph? Since this is computer graphics, the viewport is the rectangular region of the window where the image is drawn. Figure 3-16 shows a viewport that occupies most of the screen. The viewport is measured in window coordinates, which reflect the position of pixels on the screen relative to the lower-left corner of the window. Keep in mind that all vertices have been transformed by the modelview and projection matrices by this point, and vertices outside the viewing volume have been clipped.

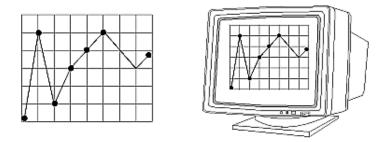


Figure 3-16 : Viewport Rectangle

Defining the Viewport

The window system, not OpenGL, is responsible for opening a window on the screen. However, by default the viewport is set to the entire pixel rectangle of the window that's opened. You use the **glViewport**() command to choose a smaller drawing region; for example, you can subdivide the window to create a split-screen effect for multiple views in the same window.

void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);

Defines a pixel rectangle in the window into which the final image is mapped. The (x, y) parameter specifies the lower-left corner of the viewport, and width and height are the size of the viewport rectangle. By default, the initial viewport values are (0, 0, winWidth, winHeight), where winWidth and winHeight are the size of the window.

The aspect ratio of a viewport should generally equal the aspect ratio of the viewing volume. If the two ratios are different, the projected image will be distorted when mapped to the viewport, as shown in Figure 3-17. Note that subsequent changes to the size of the window don't explicitly affect the viewport. Your application should detect window resize events and modify the viewport appropriately.

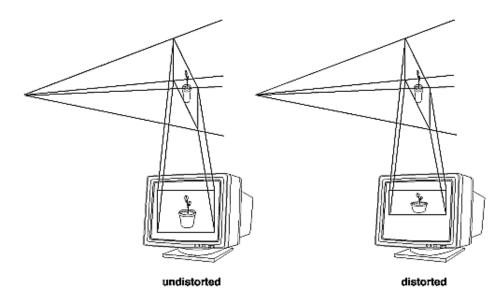


Figure 3-17 : Mapping the Viewing Volume to the Viewport

In Figure 3-17, the left figure shows a projection that maps a square image onto a square viewport using these routines:

```
gluPerspective(fovy, 1.0, near, far);
glViewport(0, 0, 400, 400);
```

However, in the right figure, the window has been resized to a nonequilateral rectangular viewport, but the projection is unchanged. The image appears compressed along the *x*-axis.

```
gluPerspective(fovy, 1.0, near, far);
glViewport (0, 0, 400, 200);
```

To avoid the distortion, modify the aspect ratio of the projection to match the viewport:

```
gluPerspective(fovy, 2.0, near, far);
glViewport(0, 0, 400, 200);
```

Try This

Modify an existing program so that an object is drawn twice, in different viewports. You might draw the

object with different projection and/or viewing transformations for each viewport. To create two side-by-side viewports, you might issue these commands, along with the appropriate modeling, viewing, and projection transformations:

The Transformed Depth Coordinate

The depth (z) coordinate is encoded during the viewport transformation (and later stored in the depth buffer). You can scale z values to lie within a desired range with the **glDepthRange**() command. (Chapter 10 discusses the depth buffer and the corresponding uses for the depth coordinate.) Unlike x and y window coordinates, z window coordinates are treated by OpenGL as though they always range from 0.0 to 1.0.

void glDepthRange(GLclampd near, GLclampd far);

Defines an encoding for z coordinates that's performed during the viewport transformation. The near and far values represent adjustments to the minimum and maximum values that can be stored in the depth buffer. By default, they're 0.0 and 1.0, respectively, which work for most applications. These parameters are clamped to lie within [0,1].

In perspective projection, the transformed depth coordinate (like the x and y coordinates) is subject to perspective division by the w coordinate. As the transformed depth coordinate moves farther away from the near clipping plane, its location becomes increasingly less precise. (See Figure 3-18.)

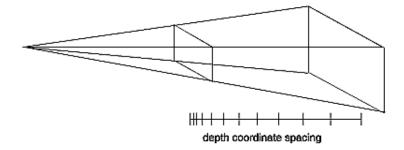


Figure 3-18 : Perspective Projection and Transformed Depth Coordinates

Therefore, perspective division affects the accuracy of operations which rely upon the transformed depth coordinate, especially depth-buffering, which is used for hidden surface removal.

Troubleshooting Transformations

It's pretty easy to get a camera pointed in the right direction, but in computer graphics, you have to specify position and direction with coordinates and angles. As we can attest, it's all too easy to achieve

the well-known black-screen effect. Although any number of things can go wrong, often you get this effect - which results in absolutely nothing being drawn in the window you open on the screen - from incorrectly aiming the "camera" and taking a picture with the model behind you. A similar problem arises if you don't choose a field of view that's wide enough to view your objects but narrow enough so they appear reasonably large.

If you find yourself exerting great programming effort only to create a black window, try these diagnostic steps.

- 1. Check the obvious possibilities. Make sure your system is plugged in. Make sure you're drawing your objects with a color that's different from the color with which you're clearing the screen. Make sure that whatever states you're using (such as lighting, texturing, alpha blending, logical operations, or antialiasing) are correctly turned on or off, as desired.
- 2. Remember that with the projection commands, the near and far coordinates measure distance from the viewpoint and that (by default) you're looking down the negative *z* axis. Thus, if the near value is 1.0 and the far 3.0, objects must have *z* coordinates between -1.0 and -3.0 in order to be visible. To ensure that you haven't clipped everything out of your scene, temporarily set the near and far clipping planes to some absurdly inclusive values, such as 0.001 and 1000000.0. This alters appearance for operations such as depth-buffering and fog, but it might uncover inadvertently clipped objects.
- 3. Determine where the viewpoint is, in which direction you're looking, and where your objects are. It might help to create a real three-dimensional space using your hands, for instance to figure these things out.
- 4. Make sure you know where you're rotating about. You might be rotating about some arbitrary location unless you translated back to the origin first. It's OK to rotate about any point unless you're expecting to rotate about the origin.
- 5. Check your aim. Use **gluLookAt**() to aim the viewing volume at your objects. Or draw your objects at or near the origin, and use **glTranslate***() as a viewing transformation to move the camera far enough in the *z* direction only so that the objects fall within the viewing volume. Once you've managed to make your objects visible, try to change the viewing volume incrementally to achieve the exact result you want, as described next.

Even after you've aimed the camera in the correct direction and you can see your objects, they might appear too small or too large. If you're using **gluPerspective(**), you might need to alter the angle defining the field of view by changing the value of the first parameter for this command. You can use trigonometry to calculate the desired field of view given the size of the object and its distance from the viewpoint: The tangent of half the desired angle is half the size of the object divided by the distance to the object (see Figure 3-19). Thus, you can use an arctangent routine to compute half the desired angle. Example 3-3 assumes such a routine, **atan2(**), which calculates the arctangent given the length of the opposite and adjacent sides of a right triangle. This result then needs to be converted from radians to degrees.

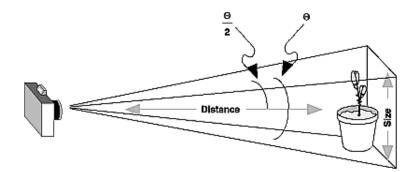


Figure 3-19 : Using Trigonometry to Calculate the Field of View

Example 3-3 : Calculating Field of View

```
#define PI 3.1415926535
double calculateAngle(double size, double distance)
{
    double radtheta, degtheta;
    radtheta = 2.0 * atan2 (size/2.0, distance);
    degtheta = (180.0 * radtheta) / PI;
    return (degtheta);
}
```

Of course, typically you don't know the exact size of an object, and the distance can only be determined between the viewpoint and a single point in your scene. To obtain a fairly good approximate value, find the bounding box for your scene by determining the maximum and minimum x, y, and z coordinates of all the objects in your scene. Then calculate the radius of a bounding sphere for that box, and use the center of the sphere to determine the distance and the radius to determine the size.

For example, suppose all the coordinates in your object satisfy the equations -1 ≤ x ≤ 3, 5 ≤ y ≤ 7, and -5 ≤ z ≤ 5. Then the center of the bounding box is (1, 6, 0), and the radius of a bounding sphere is the distance from the center of the box to any corner - say (3, 7, 5) - or

$$\sqrt{(3-1)^2 + (7-6)^2 + (5-0)^2} = \sqrt{30} = 5.477$$

If the viewpoint is at (8, 9, 10), the distance between it and the center is

$$\sqrt{(8-1)^2 + (9-6)^2 + (10-0)^2} = \sqrt{158} = 12.570$$

The tangent of the half angle is 5.477 divided by 12.570, which equals 0.4357, so the half angle is 23.54 degrees.

Remember that the field-of-view angle affects the optimal position for the viewpoint, if you're trying to achieve a realistic image. For example, if your calculations indicate that you need a 179-degree field of

view, the viewpoint must be a fraction of an inch from the screen to achieve realism. If your calculated field of view is too large, you might need to move the viewpoint farther away from the object.

Manipulating the Matrix Stacks

The modelview and projection matrices you've been creating, loading, and multiplying have only been the visible tips of their respective icebergs. Each of these matrices is actually the topmost member of a stack of matrices (see Figure 3-20).

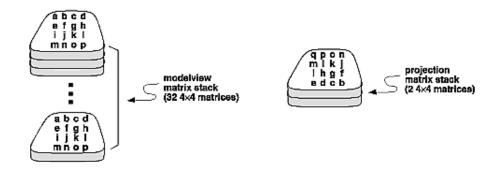


Figure 3-20 : Modelview and Projection Matrix Stacks

A stack of matrices is useful for constructing hierarchical models, in which complicated objects are constructed from simpler ones. For example, suppose you're drawing an automobile that has four wheels, each of which is attached to the car with five bolts. You have a single routine to draw a wheel and another to draw a bolt, since all the wheels and all the bolts look the same. These routines draw a wheel or a bolt in some convenient position and orientation, say centered at the origin with its axis coincident with the *z* axis. When you draw the car, including the wheels and bolts, you want to call the wheel-drawing routine four times with different transformations in effect each time to position the wheels correctly. As you draw each wheel, you want to draw the bolts five times, each time translated appropriately relative to the wheel.

Suppose for a minute that all you have to do is draw the car body and the wheels. The English description of what you want to do might be something like this:

• Draw the car body. Remember where you are, and translate to the right front wheel. Draw the wheel and throw away the last translation so your current position is back at the origin of the car body. Remember where you are, and translate to the left front wheel....

Similarly, for each wheel, you want to draw the wheel, remember where you are, and successively translate to each of the positions that bolts are drawn, throwing away the transformations after each bolt is drawn.

Since the transformations are stored as matrices, a matrix stack provides an ideal mechanism for doing this sort of successive remembering, translating, and throwing away. All the matrix operations that have been described so far (glLoadMatrix(), glMultMatrix(), glLoadIdentity() and the commands that

create specific transformation matrices) deal with the current matrix, or the top matrix on the stack. You can control which matrix is on top with the commands that perform stack operations: **glPushMatrix**(), which copies the current matrix and adds the copy to the top of the stack, and **glPopMatrix**(), which discards the top matrix on the stack, as shown in Figure 3-21. (Remember that the current matrix is always the matrix on the top.) In effect, **glPushMatrix**() means "remember where you are" and **glPopMatrix**() means "go back to where you were."

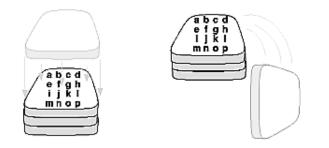


Figure 3-21 : Pushing and Popping the Matrix Stack

```
void glPushMatrix(void);
```

Pushes all matrices in the current stack down one level. The current stack is determined by **glMatrixMode()**. The topmost matrix is copied, so its contents are duplicated in both the top and second-from-the-top matrix. If too many matrices are pushed, an error is generated.

```
void glPopMatrix(void);
```

Pops the top matrix off the stack, destroying the contents of the popped matrix. What was the second-from-the-top matrix becomes the top matrix. The current stack is determined by *glMatrixMode()*. If the stack contains a single matrix, calling *glPopMatrix()* generates an error.

Example 3-4 draws an automobile, assuming the existence of routines that draw the car body, a wheel, and a bolt.

Example 3-4 : Pushing and Popping the Matrix

```
draw wheel and bolts()
{
   long i;
   draw_wheel();
   for(i=0;i<5;i++){</pre>
      glPushMatrix();
         glRotatef(72.0*i,0.0,0.0,1.0);
         glTranslatef(3.0,0.0,0.0);
         draw bolt();
      glPopMatrix();
   }
}
draw_body_and_wheel_and_bolts()
   draw_car_body();
   glPushMatrix();
                                 /*move to first wheel position*/
      glTranslatef(40,0,30);
```

```
draw_wheel_and_bolts();
glPopMatrix();
glPushMatrix();
glTranslatef(40,0,-30); /*move to 2nd wheel position*/
draw_wheel_and_bolts();
glPopMatrix();
... /*draw last two wheels similarly*/
}
```

This code assumes the wheel and bolt axes are coincident with the *z*-axis, that the bolts are evenly spaced every 72 degrees, 3 units (maybe inches) from the center of the wheel, and that the front wheels are 40 units in front of and 30 units to the right and left of the car's origin.

A stack is more efficient than an individual matrix, especially if the stack is implemented in hardware. When you push a matrix, you don't need to copy the current data back to the main process, and the hardware may be able to copy more than one element of the matrix at a time. Sometimes you might want to keep an identity matrix at the bottom of the stack so that you don't need to call **glLoadIdentity**() repeatedly.

The Modelview Matrix Stack

As you've seen earlier in "Viewing and Modeling Transformations," the modelview matrix contains the cumulative product of multiplying viewing and modeling transformation matrices. Each viewing or modeling transformation creates a new matrix that multiplies the current modelview matrix; the result, which becomes the new current matrix, represents the composite transformation. The modelview matrix stack contains at least thirty-two 4 × 4 matrices; initially, the topmost matrix is the identity matrix. Some implementations of OpenGL may support more than thirty-two matrices on the stack. To find the maximum allowable number of matrices, you can use the query command **glGetIntegerv**(GL_MAX_MODELVIEW_STACK_DEPTH, GLint **params*).

The Projection Matrix Stack

The projection matrix contains a matrix for the projection transformation, which describes the viewing volume. Generally, you don't want to compose projection matrices, so you issue **glLoadIdentity**() before performing a projection transformation. Also for this reason, the projection matrix stack need be only two levels deep; some OpenGL implementations may allow more than two 4×4 matrices. To find the stack depth, call **glGetIntegerv**(GL_MAX_PROJECTION_STACK_DEPTH, GLint **params*).

One use for a second matrix in the stack would be an application that needs to display a help window with text in it, in addition to its normal window showing a three-dimensional scene. Since text is most easily positioned with an orthographic projection, you could change temporarily to an orthographic projection, display the help, and then return to your previous projection:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix(); /*save the current projection*/
glLoadIdentity();
glOrtho(...); /*set up for displaying help*/
display_the_help();
glPopMatrix();
```

Note that you'd probably have to also change the modelview matrix appropriately.

Advanced

If you know enough mathematics, you can create custom projection matrices that perform arbitrary projective transformations. For example, the OpenGL and its Utility Library have no built-in mechanism for two-point perspective. If you were trying to emulate the drawings in drafting texts, you might need such a projection matrix.

Additional Clipping Planes

In addition to the six clipping planes of the viewing volume (left, right, bottom, top, near, and far), you can define up to six additional clipping planes to further restrict the viewing volume, as shown in Figure 3-22. This is useful for removing extraneous objects in a scene - for example, if you want to display a cutaway view of an object.

Each plane is specified by the coefficients of its equation: Ax+By+Cz+D = 0. The clipping planes are automatically transformed appropriately by modeling and viewing transformations. The clipping volume becomes the intersection of the viewing volume and all half-spaces defined by the additional clipping planes. Remember that polygons that get clipped automatically have their edges reconstructed appropriately by OpenGL.

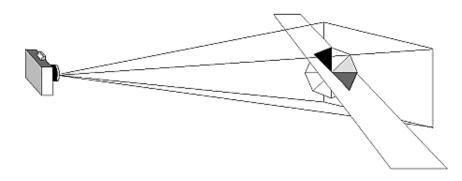


Figure 3-22 : Additional Clipping Planes and the Viewing Volume

void glClipPlane(GLenum plane, const GLdouble *equation);

Defines a clipping plane. The equation argument points to the four coefficients of the plane equation, Ax+By+Cz+D = 0. All points with eye coordinates (xe, ye, ze, we) that satisfy (A B C D)M-1 (xe ye ze we)T >= 0 lie in the half-space defined by the plane, where M is the current modelview matrix at the time **glClipPlane**() is called. All points not in this half-space are clipped away. The plane argument is GL_CLIP_PLANEi, where i is an integer specifying which of the available clipping planes to define. i is a number between 0 and one less than the maximum number of additional clipping planes.

You need to enable each additional clipping plane you define:

glEnable(GL_CLIP_PLANE*i*);

You can disable a plane with

glDisable(GL_CLIP_PLANEi);

All implementations of OpenGL must support at least six additional clipping planes, although some implementations may allow more. You can use **glGetIntegerv()** with GL_MAX_CLIP_PLANES to find how many clipping planes are supported.

Note: Clipping performed as a result of **glClipPlane**() is done in eye coordinates, not in clip coordinates. This difference is noticeable if the projection matrix is singular (that is, a real projection matrix that flattens three-dimensional coordinates to two-dimensional ones). Clipping performed in eye coordinates continues to take place in three dimensions even when the projection matrix is singular.

A Clipping Plane Code Example

Example 3-5 renders a wireframe sphere with two clipping planes that slice away three-quarters of the original sphere, as shown in Figure 3-23.



Figure 3-23 : Clipped Wireframe Sphere

Example 3-5 : Wireframe Sphere with Two Clipping Planes: clip.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}
void display(void)
{
   GLdouble eqn[4] = \{0.0, 1.0, 0.0, 0.0\};
   GLdouble eqn2[4] = \{1.0, 0.0, 0.0, 0.0\};
   glClear(GL COLOR BUFFER BIT);
   glColor3f (1.0, 1.0, 1.0);
   glPushMatrix();
   glTranslatef (0.0, 0.0, -5.0);
      clip lower half -- y < 0
/*
                                         */
   glClipPlane (GL_CLIP_PLANE0, eqn);
   glEnable (GL CLIP PLANE0);
```

```
/*
     clip left half --x < 0
                                         */
  glClipPlane (GL_CLIP_PLANE1, eqn2);
  glEnable (GL_CLIP_PLANE1);
  glRotatef (90.0, 1.0, 0.0, 0.0);
  glutWireSphere(1.0, 20, 16);
  glPopMatrix();
  glFlush ();
}
void reshape (int w, int h)
{
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL PROJECTION);
  glLoadIdentity ();
  gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
  glMatrixMode (GL MODELVIEW);
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

Try This

- Try changing the coefficients that describe the clipping planes in Example 3-5.
- Try calling a modeling transformation, such as **glRotate***(), to affect **glClipPlane**(). Make the clipping plane move independently of the objects in the scene.

Examples of Composing Several Transformations

This section demonstrates how to combine several transformations to achieve a particular result. The two examples discussed are a solar system, in which objects need to rotate on their axes as well as in orbit around each other, and a robot arm, which has several joints that effectively transform coordinate systems as they move relative to each other.

Building a Solar System

The program described in this section draws a simple solar system with a planet and a sun, both using the same sphere-drawing routine. To write this program, you need to use **glRotate***() for the revolution of the planet around the sun and for the rotation of the planet around its own axis. You also need

glTranslate*() to move the planet out to its orbit, away from the origin of the solar system. Remember that you can specify the desired size of the two spheres by supplying the appropriate arguments for the **glutWireSphere**() routine.

To draw the solar system, you first want to set up a projection and a viewing transformation. For this example, **gluPerspective()** and **gluLookAt()** are used.

Drawing the sun is straightforward, since it should be located at the origin of the grand, fixed coordinate system, which is where the sphere routine places it. Thus, drawing the sun doesn't require translation; you can use **glRotate***() to make the sun rotate about an arbitrary axis. To draw a planet rotating around the sun, as shown in Figure 3-24, requires several modeling transformations. The planet needs to rotate about its own axis once a day. And once a year, the planet completes one revolution around the sun.

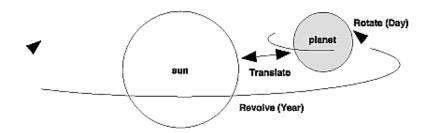


Figure 3-24 : Planet and Sun

To determine the order of modeling transformations, visualize what happens to the local coordinate system. An initial **glRotate***() rotates the local coordinate system that initially coincides with the grand coordinate system. Next, **glTranslate***() moves the local coordinate system to a position on the planet's orbit; the distance moved should equal the radius of the orbit. Thus, the initial **glRotate***() actually determines where along the orbit the planet is (or what time of year it is).

A second **glRotate***() rotates the local coordinate system around the local axes, thus determining the time of day for the planet. Once you've issued all these transformation commands, the planet can be drawn.

In summary, these are the OpenGL commands to draw the sun and planet; the full program is shown in Example 3-6.

Example 3-6 : Planetary System: planet.c

#include <GL/gl.h>
#include <GL/glu.h>

```
#include <GL/glut.h>
static int year = 0, day = 0;
void init(void)
{
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glShadeModel (GL FLAT);
}
void display(void)
{
  glClear (GL COLOR BUFFER BIT);
  glColor3f (1.0, 1.0, 1.0);
  glPushMatrix();
  glutWireSphere(1.0, 20, 16); /* draw sun */
  glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
  glTranslatef (2.0, 0.0, 0.0);
  glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
  glutWireSphere(0.2, 10, 8);
                                 /* draw smaller planet */
  glPopMatrix();
  glutSwapBuffers();
}
void reshape (int w, int h)
{
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
void keyboard (unsigned char key, int x, int y)
{
   switch (key) {
      case `d':
        day = (day + 10) % 360;
         glutPostRedisplay();
         break;
      case `D':
         day = (day - 10) % 360;
         glutPostRedisplay();
         break;
      case 'y':
         year = (year + 5) % 360;
         glutPostRedisplay();
         break;
      case `Y':
         year = (year - 5) % 360;
         glutPostRedisplay();
         break;
     default:
         break;
   }
}
int main(int argc, char** argv)
```

```
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

Try This

- Try adding a moon to the planet. Or try several moons and additional planets. Hint: Use **glPushMatrix()** and **glPopMatrix()** to save and restore the position and orientation of the coordinate system at appropriate moments. If you're going to draw several moons around a planet, you need to save the coordinate system prior to positioning each moon and restore the coordinate system after each moon is drawn.
- Try tilting the planet's axis.

Building an Articulated Robot Arm

This section discusses a program that creates an articulated robot arm with two or more segments. The arm should be connected with pivot points at the shoulder, elbow, or other joints. Figure 3-25 shows a single joint of such an arm.

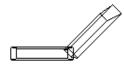


Figure 3-25 : Robot Arm

You can use a scaled cube as a segment of the robot arm, but first you must call the appropriate modeling transformations to orient each segment. Since the origin of the local coordinate system is initially at the center of the cube, you need to move the local coordinate system to one edge of the cube. Otherwise, the cube rotates about its center rather than the pivot point.

After you call **glTranslate***() to establish the pivot point and **glRotate***() to pivot the cube, translate back to the center of the cube. Then the cube is scaled (flattened and widened) before it is drawn. The **glPushMatrix**() and **glPopMatrix**() restrict the effect of **glScale***(). Here's what your code might look like for this first segment of the arm (the entire program is shown in Example 3-7):

```
glTranslatef (-1.0, 0.0, 0.0);
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
```

```
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
```

To build a second segment, you need to move the local coordinate system to the next pivot point. Since the coordinate system has previously been rotated, the *x*-axis is already oriented along the length of the rotated arm. Therefore, translating along the *x*-axis moves the local coordinate system to the next pivot point. Once it's at that pivot point, you can use the same code to draw the second segment as you used for the first one. This can be continued for an indefinite number of segments (shoulder, elbow, wrist, fingers).

```
glTranslatef (1.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
```

Example 3-7 : Robot Arm: robot.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
static int shoulder = 0, elbow = 0;
void init(void)
{
  glClearColor (0.0, 0.0, 0.0, 0.0);
 glShadeModel (GL_FLAT);
}
void display(void)
{
  glClear (GL COLOR BUFFER BIT);
  glPushMatrix();
  glTranslatef (-1.0, 0.0, 0.0);
   glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
   glTranslatef (1.0, 0.0, 0.0);
   glPushMatrix();
   glScalef (2.0, 0.4, 1.0);
   glutWireCube (1.0);
   glPopMatrix();
   glTranslatef (1.0, 0.0, 0.0);
   glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
   glTranslatef (1.0, 0.0, 0.0);
   glPushMatrix();
   glScalef (2.0, 0.4, 1.0);
   glutWireCube (1.0);
   glPopMatrix();
   glPopMatrix();
   glutSwapBuffers();
}
void reshape (int w, int h)
```

```
{
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
  glMatrixMode(GL MODELVIEW);
  glLoadIdentity();
  glTranslatef (0.0, 0.0, -5.0);
}
void keyboard (unsigned char key, int x, int y)
   switch (key) {
      case `s':
                  /* s key rotates at shoulder */
         shoulder = (shoulder + 5) % 360;
         glutPostRedisplay();
         break;
      case `S':
         shoulder = (shoulder - 5) % 360;
         glutPostRedisplay();
         break;
      case `e': /* e key rotates at elbow */
         elbow = (elbow + 5) % 360;
         glutPostRedisplay();
         break;
      case `E':
         elbow = (elbow - 5) % 360;
         glutPostRedisplay();
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
   init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

Try This

- Modify Example 3-7 to add additional segments onto the robot arm.
- Modify Example 3-7 to add additional segments at the same position. For example, give the robot arm several "fingers" at the wrist, as shown in Figure 3-26. Hint: Use **glPushMatrix()** and **glPopMatrix()** to save and restore the position and orientation of the coordinate system at the wrist. If you're going to draw fingers at the wrist, you need to save the current matrix prior to positioning each finger and restore the current matrix after each finger is drawn.

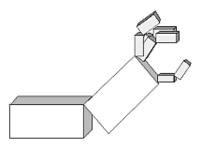


Figure 3-26 : Robot Arm with Fingers

Reversing or Mimicking Transformations

The geometric processing pipeline is very good at using viewing and projection matrices and a viewport for clipping to transform the world (or object) coordinates of a vertex into window (or screen) coordinates. However, there are situations in which you want to reverse that process. A common situation is when an application user utilizes the mouse to choose a location in three dimensions. The mouse returns only a two-dimensional value, which is the screen location of the cursor. Therefore, the application will have to reverse the transformation process to determine from where in three-dimensional space this screen location originated.

The Utility Library routine **gluUnProject**() performs this reversal of the transformations. Given the three-dimensional window coordinates for a location and all the transformations that affected them, **gluUnProject**() returns the world coordinates from where it originated.

int **gluUnProject**(GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *objx, GLdouble *objy, GLdouble *objz);

Map the specified window coordinates (winx, winy, winz) into object coordinates, using transformations defined by a modelview matrix (modelMatrix), projection matrix (projMatrix), and viewport (viewport). The resulting object coordinates are returned in objx, objy, and objz. The function returns GL_TRUE, indicating success, or GL_FALSE, indicating failure (such as an noninvertible matrix). This operation does not attempt to clip the coordinates to the viewport or eliminate depth values that fall outside of glDepthRange().

There are inherent difficulties in trying to reverse the transformation process. A two-dimensional screen location could have originated from anywhere on an entire line in three-dimensional space. To disambiguate the result, **gluUnProject()** requires that a window depth coordinate (*winz*) be provided and that *winz* be specified in terms of **glDepthRange()**. For the default values of **glDepthRange()**, *winz* at 0.0 will request the world coordinates of the transformed point at the near clipping plane, while *winz* at 1.0 will request the point at the far clipping plane.

Example 3-8 demonstrates **gluUnProject**() by reading the mouse position and determining the three-dimensional points at the near and far clipping planes from which it was transformed. The

computed world coordinates are printed to standard output, but the rendered window itself is just black.

Example 3-8 : Reversing the Geometric Processing Pipeline: unproject.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
void display(void)
   glClear(GL_COLOR_BUFFER_BIT);
   glFlush();
}
void reshape(int w, int h)
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective (45.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
   GLint viewport[4];
   GLdouble mvmatrix[16], projmatrix[16];
   GLint realy; /* OpenGL y coordinate position */
   GLdouble wx, wy, wz; /* returned world x, y, z coords
                                                            */
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN) {
            glGetIntegerv (GL_VIEWPORT, viewport);
            glGetDoublev (GL_MODELVIEW_MATRIX, mvmatrix);
            glGetDoublev (GL_PROJECTION_MATRIX, projmatrix);
   note viewport[3] is height of window in pixels
/*
                                                     */
            realy = viewport[3] - (GLint) y - 1;
            printf ("Coordinates at cursor are (%4d, %4d)\n",
               x, realy);
            gluUnProject ((GLdouble) x, (GLdouble) realy, 0.0,
               mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
            printf ("World coords at z=0.0 are (%f, %f, %f)\n",
               wx, wy, wz);
            gluUnProject ((GLdouble) x, (GLdouble) realy, 1.0,
               mvmatrix, projmatrix, viewport, &wx, &wy, &wz);
            printf ("World coords at z=1.0 are (%f, %f, %f)\n",
               wx, wy, wz);
         break;
      case GLUT_RIGHT_BUTTON:
         if (state == GLUT_DOWN)
            exit(0);
        break;
      default:
        break;
   }
```

```
}
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0;
}
```

gluProject() is another Utility Library routine, which is related to **gluUnProject**().**gluProject**() mimics the actions of the transformation pipeline. Given three-dimensional world coordinates and all the transformations that affect them, **gluProject**() returns the transformed window coordinates.

int **gluProject**(GLdouble objx, GLdouble objy, GLdouble objz, const GLdouble modelMatrix[16], const GLdouble projMatrix[16], const GLint viewport[4], GLdouble *winx, GLdouble *winy, GLdouble *winz);

Map the specified object coordinates (objx, objy, objz) into window coordinates, using transformations defined by a modelview matrix (modelMatrix), projection matrix (projMatrix), and viewport (viewport). The resulting window coordinates are returned in winx, winy, and winz. The function returns GL_TRUE, indicating success, or GL_FALSE, indicating failure.

OpenGL Programming Guide (Addison-Wesley Publishing Company)

+ +

Chapter 4 Color

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Decide between using RGBA or color-index mode for your application
- Specify desired colors for drawing objects
- Use smooth shading to draw a single polygon with more than one color

The goal of almost all OpenGL applications is to draw color pictures in a window on the screen. The window is a rectangular array of pixels, each of which contains and displays its own color. Thus, in a sense, the point of all the calculations performed by an OpenGL implementation - calculations that take into account OpenGL commands, state information, and values of parameters - is to determine the final color of every pixel that's to be drawn in the window. This chapter explains the commands for specifying colors and how OpenGL interprets them in the following major sections:

- "Color Perception" discusses how the eye perceives color.
- "Computer Color" describes the relationship between pixels on a computer *monitor* and their colors; it also defines the two display modes, RGBA and color index.
- "RGBA versus Color-Index Mode" explains how the two display modes use graphics hardware and how to decide which mode to use.
- "Specifying a Color and a Shading Model" describes the OpenGL commands you use to specify the desired color or shading model.

Color Perception

Physically, light is composed of photons - tiny particles of light, each traveling along its own path, and each vibrating at its own frequency (or wavelength, or energy - any one of frequency, wavelength, or energy determines the others). A photon is completely characterized by its position, direction, and frequency/wavelength/energy. Photons with wavelengths ranging from about 390 nanometers (nm) (violet) and 720 nm (red) cover the colors of the visible spectrum, forming the colors of a rainbow (violet, indigo, blue, green, yellow, orange, red). However, your eyes perceive lots of colors that aren't in the rainbow - white, black, brown, and pink, for example. How does this

happen?

What your eye actually sees is a mixture of photons of different frequencies. Real light sources are characterized by the distribution of photon frequencies they emit. Ideal white light consists of an equal amount of light of all frequencies. Laser light is usually very pure, and all photons have almost identical frequencies (and direction and phase, as well). Light from a sodium-vapor lamp has more light in the yellow frequency. Light from most stars in space has a distribution that depends heavily on their temperatures (black-body radiation). The frequency distribution of light from most sources in your immediate environment is more complicated.

The human eye perceives color when certain cells in the retina (called *cone cells*, or just *cones*) become excited after being struck by photons. The three different kinds of cone cells respond best to three different wavelengths of light: one type of cone cell responds best to red light, one type to green, and the other to blue. (A person who is color-blind is usually missing one or more types of cone cells.) When a given mixture of photons enters the eye, the cone cells in the retina register different degrees of excitation depending on their types, and if a different mixture of photons comes in that happens to excite the three types of cone cells to the same degrees, its color is indistinguishable from that of the first mixture.

Since each color is recorded by the eye as the levels of excitation of the cone cells by the incoming photons, the eye can perceive colors that aren't in the spectrum produced by a prism or rainbow. For example, if you send a mixture of red and blue photons so that both the red and blue cones in the retina are excited, your eye sees it as magenta, which isn't in the spectrum. Other combinations give browns, turquoises, and mauves, none of which appear in the color spectrum.

A computer-graphics monitor emulates visible colors by lighting pixels with a combination of red, green, and blue light in proportions that excite the red-, green-, and blue-sensitive cones in the retina in such a way that it matches the excitation levels generated by the photon mix it's trying to emulate. If humans had more types of cone cells, some that were yellow-sensitive for example, color monitors would probably have a yellow gun as well, and we'd use RGBY (red, green, blue, yellow) quadruples to specify colors. And if everyone were color-blind in the same way, this chapter would be simpler.

To display a particular color, the monitor sends the right amounts of red, green, and blue light to appropriately stimulate the different types of cone cells in your eye. A color monitor can send different proportions of red, green, and blue to each of the pixels, and the eye sees a million or so pinpoints of light, each with its own color.

This section considers only how the eye perceives combinations of photons that enter it. The situation for light bouncing off materials and entering the eye is even more complex - white light bouncing off a red ball will appear red, or yellow light shining through blue glass appears almost black, for example. (See "Real-World and OpenGL Lighting" in Chapter 5 for a discussion of these effects.)

Computer Color

On a color computer screen, the hardware causes each pixel on the screen to emit different amounts of red, green, and blue light. These are called the R, G, and B values. They're often packed together (sometimes with a fourth value, called alpha, or A), and the packed value is called the RGB (or RGBA) value. (See "Blending" in Chapter 6 for an explanation of the alpha values.) The color

information at each pixel can be stored either in *RGBA mode*, in which the R, G, B, and possibly A values are kept for each pixel, or in color-index mode, in which a single number (called the color index) is stored for each pixel. Each color index indicates an entry in a table that defines a particular set of R, G, and B values. Such a table is called a *color map*.

In color-index mode, you might want to alter the values in the color map. Since color maps are controlled by the window system, there are no OpenGL commands to do this. All the examples in this book initialize the color-display mode at the time the window is opened by using routines from the GLUT library. (See Appendix D for details.)

There is a great deal of variation among the different graphics hardware platforms in both the size of the pixel array and the number of colors that can be displayed at each pixel. On any graphics system, each pixel has the same amount of memory for storing its color, and all the memory for all the pixels is called the *color buffer*. The size of a buffer is usually measured in bits, so an 8-bit buffer could store 8 bits of data (256 possible different colors) for each pixel. The size of the possible buffers varies from machine to machine. (See Chapter 10 for more information.)

The R, G, and B values can range from 0.0 (none) to 1.0 (full intensity). For example, R = 0.0, G = 0.0, and B = 1.0 represents the brightest possible blue. If R, G, and B are all 0.0, the pixel is black; if all are 1.0, the pixel is drawn in the brightest white that can be displayed on the screen. *Blending* green and blue creates shades of cyan. Blue and red combine for magenta. Red and green create yellow. To help you create the colors you want from the R, G, and B components, look at the color cube shown in Plate 12. The axes of this cube represent intensities of red, blue, and green. A black-and-white version of the cube is shown in Figure 4-1.

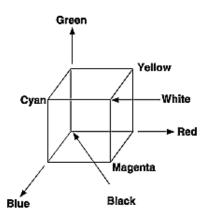


Figure 4-1 : The Color Cube in Black and White

The commands to specify a color for an object (in this case, a point) can be as simple as this:

In certain modes (for example, if lighting or texturing calculations are performed), the assigned color might go through other operations before arriving in the framebuffer as a value representing a color for a pixel. In fact, the color of a pixel is determined by a lengthy sequence of operations.

Early in a program's execution, the color-display mode is set to either RGBA mode or color-index mode. Once the color-display mode is initialized, it can't be changed. As the program executes, a color (either a color index or an RGBA value) is determined on a per-vertex basis for each geometric primitive. This color is either a color you've explicitly specified for a vertex or, if lighting is enabled, is determined from the interaction of the transformation matrices with the surface normals and other material properties. In other words, a red ball with a blue light shining on it looks different from the same ball with no light on it. (See Chapter 5 for details.) After the relevant lighting calculations are performed, the chosen shading model is applied. As explained in "Specifying a Color and a Shading Model," you can choose flat or smooth shading, each of which has different effects on the eventual color of a pixel.

Next, the primitives are *rasterized*, or converted to a two-dimensional image. Rasterizing involves determining which squares of an integer grid in window coordinates are occupied by the primitive and then assigning color and other values to each such square. A grid square along with its associated values of color, *z* (depth), and texture coordinates is called a *fragment*. Pixels are elements of the framebuffer; a fragment comes from a primitive and is combined with its corresponding pixel to yield a new pixel. Once a fragment is constructed, texturing, fog, and antialiasing are applied - if they're enabled - to the fragments. After that, any specified alpha blending, dithering, and bitwise logical operations are carried out using the fragment and the pixel already stored in the framebuffer. Finally, the fragment's color value (either color index or RGBA) is written into the pixel and displayed in the window using the window's color-display mode.

RGBA versus Color-Index Mode

In either color-index or RGBA mode, a certain amount of color data is stored at each pixel. This amount is determined by the number of bitplanes in the framebuffer. A bitplane contains 1 bit of data for each pixel. If there are 8 color bitplanes, there are 8 color bits per pixel, and hence 28 = 256 different values or colors that can be stored at the pixel.

Bitplanes are often divided evenly into storage for R, G, and B components (that is, a 24-bitplane system devotes 8 bits each to red, green, and blue), but this isn't always true. To find out the number of bitplanes available on your system for red, green, blue, alpha, or color-index values, use **glGetIntegerv**() with GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS, and GL_INDEX_BITS.

Note: Color intensities on most computer screens aren't perceived as linear by the human eye. Consider colors consisting of just a red component, with green and blue set to zero. As the intensity varies from 0.0 (off) to 1.0 (full on), the number of electrons striking the pixels increases, but the question is, does 0.5 look like halfway between 0.0 and 1.0? To test this, write a program that draws alternate pixels in a checkerboard pattern to intensities 0.0 and 1.0, and compare it with a region drawn solidly in color 0.5. From a reasonable distance from the screen, the two regions should appear to have the same intensity. If they look noticeably different, you need to use whatever correction mechanism is provided on your particular system. For example, many systems have a table to adjust intensities so that 0.5 appears to be halfway between 0.0 and 1.0. The mapping generally used is an exponential one, with the exponent referred to as gamma (hence the term gamma correction). Using the same gamma for the red, green, and blue components gives pretty good results, but three different gamma values might give slightly better results. (For more details on this topic, see Foley, van Dam, et al. *Computer Graphics: Principles and Practice*. Reading, MA: Addison-Wesley Developers Press, 1990.)

RGBA Display Mode

In RGBA mode, the hardware sets aside a certain number of bitplanes for each of the R, G, B, and A components (not necessarily the same number for each component) as shown in Figure 4-2. The R, G, and B values are typically stored as integers rather than floating-point numbers, and they're scaled to the number of available bits for storage and retrieval. For example, if a system has 8 bits available for the R component, integers between 0 and 255 can be stored; thus, 0, 1, 2, ..., 255 in the bitplanes would correspond to R values of 0/255 = 0.0, 1/255, 2/255, ..., 255/255 = 1.0. Regardless of the number of bitplanes, 0.0 specifies the minimum intensity, and 1.0 specifies the maximum intensity.

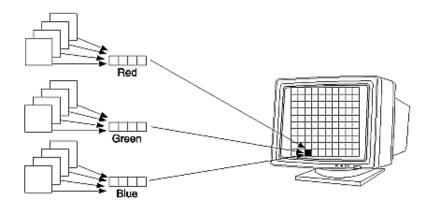


Figure 4-2 : RGB Values from the Bitplanes

Note: The alpha value (the A in RGBA) has no direct effect on the color displayed on the screen. It can be used for many things, including blending and transparency, and it can have an effect on the values of R, G, and B that are written. (See "Blending" in Chapter 6 for more information about alpha values.)

The number of distinct colors that can be displayed at a single pixel depends on the number of bitplanes and the capacity of the hardware to interpret those bitplanes. The number of distinct colors can't exceed 2n, where n is the number of bitplanes. Thus, a machine with 24 bitplanes for RGB can display up to 16.77 million distinct colors.

Dithering

Advanced

Some graphics hardware uses dithering to increase the number of apparent colors. Dithering is the technique of using combinations of some colors to create the effect of other colors. To illustrate how dithering works, suppose your system has only 1 bit each for R, G, and B and thus can display only eight colors: black, white, red, blue, green, yellow, cyan, and magenta. To display a pink region, the hardware can fill the region in a checkerboard manner, alternating red and white pixels. If your eye is far enough away from the screen that it can't distinguish individual pixels, the region appears pink - the average of red and white. Redder pinks can be achieved by filling a higher proportion of the pixels with red, whiter pinks would use more white pixels, and so on.

With this technique, there are no pink pixels. The only way to achieve the effect of "pinkness" is to

cover a region consisting of multiple pixels - you can't dither a single pixel. If you specify an RGB value for an unavailable color and fill a polygon, the hardware fills the pixels in the interior of the polygon with a mixture of nearby colors whose average appears to your eye to be the color you want. (Remember, though, that if you're reading pixel information out of the framebuffer, you get the actual red and white pixel values, since there aren't any pink ones. See Chapter 8 for more information about reading pixel values.)

Figure 4-3 illustrates some simple dithering of black and white pixels to make shades of gray. From left to right, the 4×4 patterns at the top represent dithering patterns for 50 percent, 19 percent, and 69 percent gray. Under each pattern, you can see repeated reduced copies of each pattern, but these black and white squares are still bigger than most pixels. If you look at them from across the room, you can see that they blur together and appear as three levels of gray.

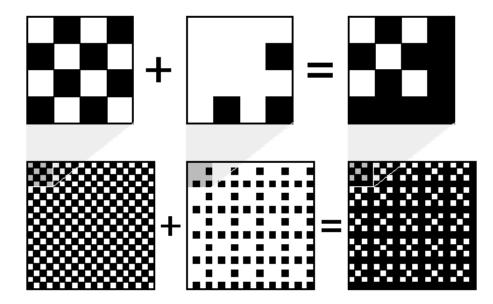


Figure 4-3 : Dithering Black and White to Create Gray

With about 8 bits each of R, G, and B, you can get a fairly high-quality image without dithering. Just because your machine has 24 color bitplanes, however, doesn't mean that dithering won't be desirable. For example, if you are running in double-buffer mode, the bitplanes might be divided into two sets of twelve, so there are really only 4 bits each per R, G, and B component. Without dithering, 4-bit-per-component color can give less than satisfactory results in many situations.

You enable or disable dithering by passing GL_DITHER to **glEnable**() or **glDisable**(). Note that dithering, unlike many other features, is enabled by default.

Color-Index Display Mode

With color-index mode, OpenGL uses a color map (or *lookup table*), which is similar to using a palette to mix paints to prepare for a paint-by-number scene. A painter's palette provides spaces to mix paints together; similarly, a computer's color map provides indices where the primary red, green, and blue values can be mixed, as shown in Figure 4-4.

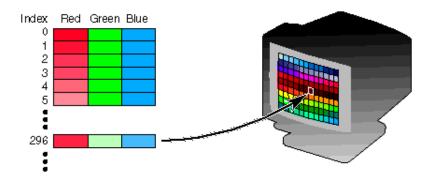


Figure 4-4 : A Color Map

A painter filling in a paint-by-number scene chooses a color from the color palette and fills the corresponding numbered regions with that color. A computer stores the color index in the bitplanes for each pixel. Then those bitplane values reference the color map, and the screen is painted with the corresponding red, green, and blue values from the color map, as shown in Figure 4-5.

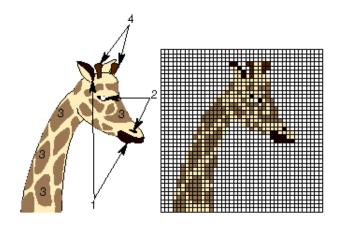


Figure 4-5 : Using a Color Map to Paint a Picture

In color-index mode, the number of simultaneously available colors is limited by the size of the color map and the number of bitplanes available. The size of the color map is determined by the amount of hardware dedicated to it. The size of the color map is always a power of 2, and typical sizes range from 256 (28) to 4096 (212), where the exponent is the number of bitplanes being used. If there are 2n indices in the color map and m available bitplanes, the number of usable entries is the smaller of 2n and 2m.

With RGBA mode, each pixel's color is independent of other pixels. However, in color-index mode, each pixel with the same index stored in its bitplanes shares the same color-map location. If the contents of an entry in the color map change, then all pixels of that color index change their color.

Choosing between RGBA and Color-Index Mode

You should base your decision to use RGBA or color-index mode on what hardware is available and on what your application needs. For most systems, more colors can be simultaneously

represented with RGBA mode than with color-index mode. Also, for several effects, such as shading, lighting, texture mapping, and fog, RGBA provides more flexibility than color-index mode.

You might prefer to use color-index mode in the following cases:

- If you're porting an existing application that makes significant use of color-index mode, it might be easier to not change to RGBA mode.
- If you have a small number of bitplanes available, RGBA mode may produce noticeably coarse shades of colors. For example, if you have only 8 bitplanes, in RGBA mode, you may have only 3 bits for red, 3 bits for green, and 2 bits for blue. You'd only have 8 (23) shades of red and green, and only 4 shades of blue. The gradients between color shades are likely to be very obvious.

In this situation, if you have limited shading requirements, you can use the color lookup table to load more shades of colors. For example, if you need only shades of blue, you can use color-index mode and store up to 256 (28) shades of blue in the color-lookup table, which is much better than the 4 shades you would have in RGBA mode. Of course, this example would use up your entire color-lookup table, so you would have no shades of red, green, or other combined colors.

• Color-index mode can be useful for various tricks, such as color-map animation and drawing in layers. (See Chapter 14 for more information.)

In general, use RGBA mode wherever possible. It works with texture mapping and works better with lighting, shading, fog, antialiasing, and blending.

Changing between Display Modes

In the best of all possible worlds, you might want to avoid making a choice between RGBA and color-index display mode. For example, you may want to use color-index mode for a color-map animation effect and then, when needed, immediately change the scene to RGBA mode for texture mapping.

Or similarly, you may desire to switch between single and double buffering. For example, you may have very few bitplanes; let's say 8 bitplanes. In single-buffer mode, you'll have 256 (28) colors, but if you are using double-buffer mode to eliminate flickering from your animated program, you may only have 16 (24) colors. Perhaps you want to draw a moving object without flicker and are willing to sacrifice colors for using double-buffer mode (maybe the object is moving so fast that the viewer won't notice the details). But when the object comes to rest, you will want to draw it in single-buffer mode so that you can use more colors.

Unfortunately, most window systems won't allow an easy switch. For example, with the X Window System, the color-display mode is an attribute of the X Visual. An X Visual must be specified before the window is created. Once it is specified, it cannot be changed for the life of the window. After you create a window with a double-buffered, RGBA display mode, you're stuck with it.

A tricky solution to this problem is to create more than one window, each with a different display mode. Then you must control the visibility of the windows (for example, mapping or unmapping an X Window, or managing or unmanaging a Motif or Athena widget) and draw the object into the

Specifying a Color and a Shading Model

OpenGL maintains a current color (in RGBA mode) and a current color index (in color-index mode). Unless you're using a more complicated coloring model such as lighting or texture mapping, each object is drawn using the current color (or color index). Look at the following pseudocode sequence:

```
set_color(RED);
draw_item(A);
draw_item(B);
set_color(GREEN);
set_color(BLUE);
draw_item(C);
```

Items A and B are drawn in red, and item C is drawn in blue. The fourth line, which sets the current color to green, has no effect (except to waste a bit of time). With no lighting or texturing, when the current color is set, all items drawn afterward are drawn in that color until the current color is changed to something else.

Specifying a Color in RGBA Mode

In RGBA mode, use the **glColor***() command to select a current color.

```
void glColor3{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb);
void glColor4{b s i f d ub us ui} (TYPEr, TYPEg, TYPEb, TYPEa);
void glColor3{b s i f d ub us ui}v (const TYPE*v);
void glColor4{b s i f d ub us ui}v (const TYPE*v);
Sets the current red_green_blue_and glpha values_This comma
```

Sets the current red, green, blue, and alpha values. This command can have up to three suffixes, which differentiate variations of the parameters accepted. The first suffix is either 3 or 4, to indicate whether you supply an alpha value in addition to the red, green, and blue values. If you don't supply an alpha value, it's automatically set to 1.0. The second suffix indicates the data type for parameters: byte, short, integer, float, double, unsigned byte, unsigned short, or unsigned integer. The third suffix is an optional v, which indicates that the argument is a pointer to an array of values of the given data type.

For the versions of **glColor***() that accept floating-point data types, the values should typically range between 0.0 and 1.0, the minimum and maximum values that can be stored in the framebuffer. Unsigned-integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and zero maps to 0.0 (zero intensity). Signed-integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0 (see Table 4-1).

Neither floating-point nor signed-integer values are clamped to the range [0,1] before updating the current color or current lighting material parameters. After lighting calculations, resulting color values outside the range [0,1] are clamped to the range [0,1] before they are interpolated or written into a color buffer. Even if lighting is disabled, the color components are clamped before rasterization.

Suffix	Data Type	Minimum Value	Min Value Maps to	Maximum Value	Max Value Maps to
b	1-byte integer	-128	-1.0	127	1.0
8	2-byte integer	-32,768	-1.0	32,767	1.0
i	4-byte integer	-2,147,483,648	-1.0	2,147,483,647	1.0
ub	unsigned 1-byte integer	0	0.0	255	1.0
us	unsigned 2-byte integer	0	0.0	65,535	1.0
ui	unsigned 4-byte integer	0	0.0	4,294,967,295	1.0

Table 4-1 : Converting Color Values to Floating-Point Numbers

Specifying a Color in Color-Index Mode

In color-index mode, use the **glIndex*()** command to select a single-valued color index as the current color index.

void glIndex{sifd ub}(TYPE c);

void glIndex{sifd ub}v(const TYPE *c);

Sets the current color index to c. The first suffix for this command indicates the data type for parameters: short, integer, float, double, or unsigned byte. The second, optional suffix is *v*, which indicates that the argument is an array of values of the given data type (the array contains only one value).

In "Clearing the Window" in Chapter 2, you saw the specification of **glClearColor**(). For color-index mode, there is a corresponding **glClearIndex**().

void glClearIndex(GLfloat cindex);

Sets the current clearing color in color-index mode. In a color-index mode window, a call to *glClear*(*GL_COLOR_BUFFER_BIT*) will use cindex to clear the buffer. The default clearing index is 0.0.

Note: OpenGL does not have any routines to load values into the color-lookup table. Window systems typically already have such operations. GLUT has the routine **glutSetColor()** to call the window-system specific commands.

Advanced

The current index is stored as a floating-point value. Integer values are converted directly to floating-point values, with no special mapping. Index values outside the representable range of the color-index buffer aren't clamped. However, before an index is dithered (if enabled) and written to the framebuffer, it's converted to fixed-point format. Any bits in the integer portion of the resulting fixed-point value that don't correspond to bits in the framebuffer are masked out.

Specifying a Shading Model

A line or a filled polygon primitive can be drawn with a single color (flat shading) or with many different colors (smooth shading, also called Gouraud shading). You specify the desired shading technique with **glShadeModel**().

void glShadeModel (GLenum mode);

Sets the shading model. The mode parameter can be either GL_SMOOTH (the default) or GL_FLAT.

With flat shading, the color of one particular vertex of an independent primitive is duplicated across all the primitive's vertices to render that primitive. With smooth shading, the color at each vertex is treated individually. For a line primitive, the colors along the line segment are interpolated between the vertex colors. For a polygon primitive, the colors for the interior of the polygon are interpolated between the vertex colors. Example 4-1 draws a smooth-shaded triangle, as shown in "Plate 11" in Appendix I.

Example 4-1 : Drawing a Smooth-Shaded Triangle: smooth.c

```
#include <GL/gl.h>
#include <GL/glut.h>
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
}
void triangle(void)
{
   glBegin (GL_TRIANGLES);
   glColor3f (1.0, 0.0, 0.0);
   glVertex2f (5.0, 5.0);
   glColor3f (0.0, 1.0, 0.0);
   glVertex2f (25.0, 5.0);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2f (5.0, 25.0);
   glEnd();
}
void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   triangle ();
   glFlush ();
}
void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
```

```
if (w \le h)
      gluOrtho2D (0.0, 30.0, 0.0, 30.0*(GLfloat) h/(GLfloat) w);
  else
     gluOrtho2D (0.0, 30.0*(GLfloat) w/(GLfloat) h, 0.0, 30.0);
  glMatrixMode(GL_MODELVIEW);
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT SINGLE | GLUT RGB);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

With smooth shading, neighboring pixels have slightly different color values. In RGBA mode, adjacent pixels with slightly different values look similar, so the color changes across a polygon appear gradual. In color-index mode, adjacent pixels may reference different locations in the color-index table, which may not have similar colors at all. Adjacent color-index entries may contain wildly different colors, so a smooth-shaded polygon in color-index mode can look psychedelic.

To avoid this problem, you have to create a color ramp of smoothly changing colors among a contiguous set of indices in the color map. Remember that loading colors into a color map is performed through your window system rather than OpenGL. If you use GLUT, you can use **glutSetColor**() to load a single index in the color map with specified red, green, and blue values. The first argument for **glutSetColor**() is the index, and the others are the red, green, and blue values. To load thirty-two contiguous color indices (from color index 16 to 47) with slightly differing shades of yellow, you might call

```
for (i = 0; i < 32; i++) {
  glutSetColor (16+i, 1.0*(i/32.0), 1.0*(i/32.0), 0.0);
}</pre>
```

Now, if you render smooth-shaded polygons that use only the colors from index 16 to 47, those polygons have gradually differing shades of yellow.

With flat shading, the color of a single vertex defines the color of an entire primitive. For a line segment, the color of the line is the current color when the second (ending) vertex is specified. For a polygon, the color used is the one that's in effect when a particular vertex is specified, as shown in Table 4-2. The table counts vertices and polygons starting from 1. OpenGL follows these rules consistently, but the best way to avoid uncertainty about how a flat-shaded primitive will be drawn is to specify only one color for the primitive.

Table 4-2 : How OpenGL Selects a Color for the ith Flat-Shaded Polygon

Type of Polygon	Vertex Used to Select the Color for the ith Polygon
single polygon	1
triangle strip	i+2
triangle fan	i+2
independent triangle	3i
quad strip	2i+2
independent quad	4i

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

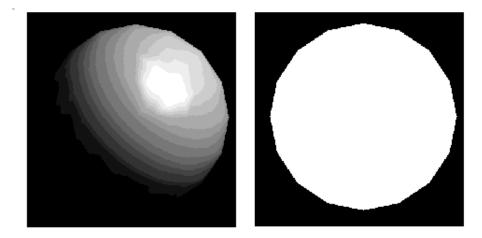
Chapter 5 Lighting

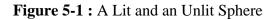
Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand how real-world lighting conditions are approximated by OpenGL
- Render illuminated objects by defining the desired light sources and lighting model
- Define the material properties of the objects being illuminated
- Manipulate the matrix stack to control the position of light sources

As you saw in Chapter 4, OpenGL computes the color of each pixel in a final, displayed scene that's held in the framebuffer. Part of this computation depends on what lighting is used in the scene and on how objects in the scene reflect or absorb that light. As an example of this, recall that the ocean has a different color on a bright, sunny day than it does on a gray, cloudy day. The presence of sunlight or clouds determines whether you see the ocean as bright turquoise or murky gray-green. In fact, most objects don't even look three-dimensional until they're lit. Figure 5-1 shows two versions of the exact same scene (a single sphere), one with lighting and one without.





As you can see, an unlit sphere looks no different from a two-dimensional disk. This demonstrates how critical the interaction between objects and light is in creating a three-dimensional scene.

With OpenGL, you can manipulate the lighting and objects in a scene to create many different

kinds of effects. This chapter begins with a primer on hidden-surface removal. Then it explains how to control the lighting in a scene, discusses the OpenGL conceptual model of lighting, and describes in detail how to set the numerous illumination parameters to achieve certain effects. Toward the end of the chapter, the mathematical computations that determine how lighting affects color are presented.

This chapter contains the following major sections:

- "A Hidden-Surface Removal Survival Kit" describes the basics of removing hidden surfaces from view.
- "Real-World and OpenGL Lighting" explains in general terms how light behaves in the world and how OpenGL models this behavior.
- "A Simple Example: Rendering a Lit Sphere" introduces the OpenGL lighting facility by presenting a short program that renders a lit sphere.
- "Creating Light Sources" explains how to define and position light sources.
- "Selecting a Lighting Model" discusses the elements of a lighting model and how to specify them.
- "Defining Material Properties" explains how to describe the properties of objects so that they interact with light in a desired way.
- "The Mathematics of Lighting" presents the mathematical calculations used by OpenGL to determine the effect of lights in a scene.
- "Lighting in Color-Index Mode" discusses the differences between using RGBA mode and color-index mode for lighting.

A Hidden-Surface Removal Survival Kit

With this section, you begin to draw shaded, three-dimensional objects, in earnest. With shaded polygons, it becomes very important to draw the objects that are closer to our viewing position and to eliminate objects obscured by others nearer to the eye.

When you draw a scene composed of three-dimensional objects, some of them might obscure all or parts of others. Changing your viewpoint can change the obscuring relationship. For example, if you view the scene from the opposite direction, any object that was previously in front of another is now behind it. To draw a realistic scene, these obscuring relationships must be maintained. Suppose your code works like this:

```
while (1) {
   get_viewing_point_from_mouse_position();
   glClear(GL_COLOR_BUFFER_BIT);
   draw_3d_object_A();
   draw_3d_object_B();
}
```

For some mouse positions, object A might obscure object B. For others, the reverse may hold. If nothing special is done, the preceding code always draws object B second (and thus on top of object A) no matter what viewing position is selected. In a worst case scenario, if objects A and B intersect one another so that part of object A obscures object B and part of B obscures A, changing the drawing order does not provide a solution.

The elimination of parts of solid objects that are obscured by others is called *hidden-surface removal*. (Hidden-line removal, which does the same job for objects represented as wireframe skeletons, is a bit trickier and isn't discussed here. See "Hidden-Line Removal" in Chapter 14 for details.) The easiest way to achieve hidden-surface removal is to use the depth buffer (sometimes called a z-buffer). (Also see Chapter 10.)

A depth buffer works by associating a depth, or distance, from the view plane (usually the near clipping plane), with each pixel on the window. Initially, the depth values for all pixels are set to the largest possible distance (usually the far clipping plane) using the **glClear**() command with GL_DEPTH_BUFFER_BIT. Then the objects in the scene are drawn in any order.

Graphical calculations in hardware or software convert each surface that's drawn to a set of pixels on the window where the surface will appear if it isn't obscured by something else. In addition, the distance from the view plane is computed. With depth buffering enabled, before each pixel is drawn a comparison is done with the depth value already stored at the pixel. If the new pixel is closer than (in front of) what's there, the new pixel's color and depth values replace those that are currently written into the pixel. If the new pixel's depth is greater than what's currently there, the new pixel is obscured, and the color and depth information for the incoming pixel is discarded.

To use depth buffering, you need to enable depth buffering. This has to be done only once. Before drawing, each time you draw the scene, you need to clear the depth buffer and then draw the objects in the scene in any order.

To convert the preceding code example so that it performs hidden-surface removal, modify it to the following:

```
glutInitDisplayMode (GLUT_DEPTH | .... );
glEnable(GL_DEPTH_TEST);
...
while (1) {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   get_viewing_point_from_mouse_position();
   draw_3d_object_A();
   draw_3d_object_B();
}
```

The argument to **glClear()** clears both the depth and color buffers.

Depth-buffer testing can affect the performance of your application. Since information is discarded rather than used for drawing, hidden-surface removal can increase your performance slightly. However, the implementation of your depth buffer probably has the greatest effect on performance. A "software" depth buffer (implemented with processor memory) may be much slower than one implemented with a specialized hardware depth buffer.

Real-World and OpenGL Lighting

When you look at a physical surface, your eye's perception of the color depends on the distribution of photon energies that arrive and trigger your cone cells. (See "Color Perception" in Chapter 4.) Those photons come from a light source or combination of sources, some of which are absorbed and some of which are reflected by the surface. In addition, different surfaces may have very different properties - some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions. Most surfaces are somewhere in between.

OpenGL approximates light and lighting as if light can be broken into red, green, and blue components. Thus, the color of light sources is characterized by the amount of red, green, and blue light they emit, and the material of surfaces is characterized by the percentage of the incoming red, green, and blue components that is reflected in various directions. The OpenGL lighting equations are just an approximation but one that works fairly well and can be computed relatively quickly. If you desire a more accurate (or just different) lighting model, you have to do your own calculations in software. Such software can be enormously complex, as a few hours of reading any optics textbook should convince you.

In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on and off. Some light comes from a particular direction or position, and some light is generally scattered about the scene. For example, when you turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one, two, three, or more walls. This bounced light (called ambient) is assumed to be so scattered that there is no way to tell its original direction, but it disappears if a particular light source is turned off.

Finally, there might be a general ambient light in the scene that comes from no particular source, as if it had been scattered so many times that its original source is impossible to determine.

In the OpenGL model, the light sources have an effect only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material might emit its own light (like headlights on an automobile), it might scatter some incoming light in all directions, and it might reflect some portion of the incoming light in a preferential direction like a mirror or other shiny surface.

The OpenGL lighting model considers the lighting to be divided into four independent components: emissive, ambient, diffuse, and specular. All four components are computed independently and then added together.

Ambient, Diffuse, and Specular Light

Ambient illumination is light that's been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions. Backlighting in a room has a large ambient component, since most of the light that reaches your eye has first bounced off many surfaces. A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since you're outdoors, very little of the light reaches your eye after bouncing off other objects. When ambient light strikes a surface, it's scattered equally in all directions.

The diffuse component is the light that comes from one direction, so it's brighter if it comes squarely down on a surface than if it barely glances off the surface. Once it hits a surface, however, it's scattered equally in all directions, so it appears equally bright, no matter where the eye is located. Any light coming from a particular position or direction probably has a diffuse component.

Finally, specular light comes from a particular direction, and it tends to bounce off the surface in a

preferred direction. A well-collimated laser beam bouncing off a high-quality mirror produces almost 100 percent specular reflection. Shiny metal or plastic has a high specular component, and chalk or carpet has almost none. You can think of specularity as shininess.

Although a light source delivers a single distribution of frequencies, the ambient, diffuse, and specular components might be different. For example, if you have a white light in a room with red walls, the scattered light tends to be red, although the light directly striking objects is white. OpenGL allows you to set the red, green, and blue values for each component of light independently.

Material Colors

The OpenGL lighting model makes the approximation that a material's color depends on the percentages of the incoming red, green, and blue light it reflects. For example, a perfectly red ball reflects all the incoming red light and absorbs all the green and blue light that strikes it. If you view such a ball in white light (composed of equal amounts of red, green, and blue light), all the red is reflected, and you see a red ball. If the ball is viewed in pure red light, it also appears to be red. If, however, the red ball is viewed in pure green light, it appears black (all the green is absorbed, and there's no incoming red, so no light is reflected).

Like lights, materials have different ambient, diffuse, and specular colors, which determine the ambient, diffuse, and specular reflectances of the material. A material's ambient reflectance is combined with the ambient component of each incoming light source, the diffuse reflectance with the light's diffuse component, and similarly for the specular reflectance and component. Ambient and diffuse reflectances define the color of the material and are typically similar if not identical. Specular reflectance is usually white or gray, so that specular highlights end up being the color of the light source's specular intensity. If you think of a white light shining on a shiny red plastic sphere, most of the sphere appears red, but the shiny highlight is white.

In addition to ambient, diffuse, and specular colors, materials have an *emissive* color, which simulates light originating from an object. In the OpenGL lighting model, the emissive color of a surface adds intensity to the object, but is unaffected by any light sources. Also, the emissive color does not introduce any additional light into the overall scene.

RGB Values for Lights and Materials

The color components specified for lights mean something different than for materials. For a light, the numbers correspond to a percentage of full intensity for each color. If the R, G, and B values for a light's color are all 1.0, the light is the brightest possible white. If the values are 0.5, the color is still white, but only at half intensity, so it appears gray. If R=G=1 and B=0 (full red and green with no blue), the light appears yellow.

For materials, the numbers correspond to the reflected proportions of those colors. So if R=1, G=0.5, and B=0 for a material, that material reflects all the incoming red light, half the incoming green, and none of the incoming blue light. In other words, if an OpenGL light has components (LR, LG, LB), and a material has corresponding components (MR, MG, MB), then, ignoring all other reflectivity effects, the light that arrives at the eye is given by (LR*MR, LG*MG, LB*MB).

Similarly, if you have two lights that send (R1, G1, B1) and (R2, G2, B2) to the eye, OpenGL adds the components, giving (R1+R2, G1+G2, B1+B2). If any of the sums are greater than 1 (corresponding to a color brighter than the equipment can display), the component is clamped to 1.

A Simple Example: Rendering a Lit Sphere

These are the steps required to add lighting to your scene.

- 1. Define normal vectors for each vertex of all the objects. These normals determine the orientation of the object relative to the light sources.
- 2. Create, select, and position one or more light sources.
- 3. Create and select a *lighting model*, which defines the level of global ambient light and the effective location of the viewpoint (for the purposes of lighting calculations).
- 4. Define material properties for the objects in the scene.

Example 5-1 accomplishes these tasks. It displays a sphere illuminated by a single light source, as shown earlier in Figure 5-1.

Example 5-1 : Drawing a Lit Sphere: light.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void)
ł
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 50.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_SMOOTH);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}
void display(void)
   glClear (GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
   glutSolidSphere (1.0, 20, 16);
   glFlush ();
}
void reshape (int w, int h)
ł
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity();
   if (w \le h)
      glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
         1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
```

```
1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
   glMatrixMode(GL MODELVIEW);
   glLoadIdentity();
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT RGB | GLUT DEPTH);
  glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

The lighting-related calls are in the **init**() command; they're discussed briefly in the following paragraphs and in more detail later in the chapter. One thing to note about Example 5-1 is that it uses RGBA color mode, not color-index mode. The OpenGL lighting calculation is different for the two modes, and in fact the lighting capabilities are more limited in color-index mode. Thus, RGBA is the preferred mode when doing lighting, and all the examples in this chapter use it. (See "Lighting in Color-Index Mode" for more information about lighting in color-index mode.)

Define Normal Vectors for Each Vertex of Every Object

An object's normals determine its orientation relative to the light sources. For each vertex, OpenGL uses the assigned normal to determine how much light that particular vertex receives from each light source. In this example, the normals for the sphere are defined as part of the **glutSolidSphere()** routine. (See "Normal Vectors" in Chapter 2 for more details on how to define normals.)

Create, Position, and Enable One or More Light Sources

Example 5-1 uses only one, white light source; its location is specified by the **glLightfv**() call. This example uses the default color for light zero (GL_LIGHT0), which is white; if you want a differently colored light, use **glLight***() to indicate this. You can include at least eight different light sources in your scene of various colors; the default color of these other lights is black. (The particular implementation of OpenGL you're using might allow more than eight.) You can also locate the lights wherever you desire - you can position them near the scene, as a desk lamp would be, or an infinite distance away, like the sun. In addition, you can control whether a light produces a narrow, focused beam or a wider beam. Remember that each light source adds significantly to the calculations needed to render the scene, so performance is affected by the number of lights in the scene. (See "Creating Light Sources" for more information about how to create lights with the desired characteristics.)

After you've defined the characteristics of the lights you want, you have to turn them on with the **glEnable**() command. You also need to call **glEnable**() with GL_LIGHTING as a parameter to prepare OpenGL to perform lighting calculations. (See "Enabling Lighting" for more information.)

Select a Lighting Model

As you might expect, the **glLightModel***() command describes the parameters of a lighting model.

In Example 5-1, the only element of the lighting model that's defined explicitly is the global ambient light. The lighting model also defines whether the viewer of the scene should be considered to be an infinite distance away or local to the scene, and whether lighting calculations should be performed differently for the front and back surfaces of objects in the scene. Example 5-1 uses the default settings for these two aspects of the model - an infinite viewer and one-sided lighting. Using a local viewer adds significantly to the complexity of the calculations that must be performed, because OpenGL must calculate the angle between the viewpoint and each object. With an infinite viewer, however, the angle is ignored, and the results are slightly less realistic. Further, since in this example, the back surface of the sphere is never seen (it's the inside of the sphere), one-sided lighting is sufficient. (See "Selecting a Lighting Model" for a more detailed description of the elements of an OpenGL lighting model.)

Define Material Properties for the Objects in the Scene

An object's material properties determine how it reflects light and therefore what material it seems to be made of. Because the interaction between an object's material surface and incident light is complex, specifying material properties so that an object has a certain desired appearance is an art. You can specify a material's ambient, diffuse, and specular colors and how shiny it is. In this example, only these last two material properties - the specular material color and shininess - are explicitly specified (with the **glMaterialfv**() calls). (See "Defining Material Properties" for a description and examples of all the material-property parameters.)

Some Important Notes

As you write your own lighting program, remember that you can use the default values for some lighting parameters; others need to be changed. Also, don't forget to enable whatever lights you define and to enable lighting calculations. Finally, remember that you might be able to use display lists to maximize efficiency as you change lighting conditions. (See "Display-List Design Philosophy" in Chapter 7.)

Creating Light Sources

Light sources have a number of properties, such as color, position, and direction. The following sections explain how to control these properties and what the resulting light looks like. The command used to specify all properties of lights is **glLight***(); it takes three arguments: to identify the light whose property is being specified, the property, and the desired value for that property.

void glLight{if}(GLenum light, GLenum pname, TYPEparam);

void glLight{if}v(GLenum light, GLenum pname, TYPE *param);

Creates the light specified by light, which can be GL_LIGHT0, GL_LIGHT1, ..., or GL_LIGHT7. The characteristic of the light being set is defined by pname, which specifies a named parameter (see Table 5-1). param indicates the values to which the pname characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued light characteristics.

 Table 5-1 : Default Values for pname Parameter of glLight*()

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(x, y, z) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

Note: The default values listed for GL_DIFFUSE and GL_SPECULAR in Table 5-1 apply only to GL_LIGHT0. For other lights, the default value is (0.0, 0.0, 0.0, 1.0) for both GL_DIFFUSE and GL_SPECULAR.

Example 5-2 shows how to use **glLight***():

Example 5-2 : Defining Colors and Position for a Light Source

GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 }; GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 }; GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 }; GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 }; glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient); glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse); glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular); glLightfv(GL_LIGHT0, GL_POSITION, light_position);

As you can see, arrays are defined for the parameter values, and **glLightfv**() is called repeatedly to set the various parameters. In this example, the first three calls to **glLightfv**() are superfluous, since they're being used to specify the default values for the GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR parameters.

Note: Remember to turn on each light with **glEnable**(). (See "Enabling Lighting" for more information about how to do this.)

All the parameters for **glLight***() and their possible values are explained in the following sections. These parameters interact with those that define the overall lighting model for a particular scene and an object's material properties. (See "Selecting a Lighting Model" and "Defining Material Properties" for more information about these two topics. "The Mathematics of Lighting" explains how all these parameters interact mathematically.)

Color

OpenGL allows you to associate three different color-related parameters - GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR - with any particular light. The GL_AMBIENT parameter refers to the RGBA intensity of the ambient light that a particular light source adds to the scene. As you can see in Table 5-1, by default there is no ambient light since GL_AMBIENT is (0.0, 0.0, 0.0, 1.0). This value was used in Example 5-1. If this program had specified blue ambient light as

GLfloat light_ambient[] = { 0.0, 0.0, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);

the result would have been as shown in the left side of "Plate 13" in Appendix I.

The GL_DIFFUSE parameter probably most closely correlates with what you naturally think of as "the color of a light." It defines the RGBA color of the diffuse light that a particular light source adds to a scene. By default, GL_DIFFUSE is (1.0, 1.0, 1.0, 1.0) for GL_LIGHT0, which produces a bright, white light as shown in the left side of "Plate 13" in Appendix I. The default value for any other light (GL_LIGHT1, ..., GL_LIGHT7) is (0.0, 0.0, 0.0, 0.0).

The GL_SPECULAR parameter affects the color of the specular highlight on an object. Typically, a real-world object such as a glass bottle has a specular highlight that's the color of the light shining on it (which is often white). Therefore, if you want to create a realistic effect, set the GL_SPECULAR parameter to the same value as the GL_DIFFUSE parameter. By default, GL_SPECULAR is (1.0, 1.0, 1.0, 1.0) for GL_LIGHT0 and (0.0, 0.0, 0.0, 0.0) for any other light.

Note: The alpha component of these colors is not used until blending is enabled. (See Chapter 6.) Until then, the alpha value can be safely ignored.

Position and Attenuation

As previously mentioned, you can choose whether to have a light source that's treated as though it's located infinitely far away from the scene or one that's nearer to the scene. The first type is referred to as a *directional* light source; the effect of an infinite location is that the rays of light can be considered parallel by the time they reach an object. An example of a real-world directional light source is the sun. The second type is called a *positional* light source, since its exact position within the scene determines the effect it has on a scene and, specifically, the direction from which the light rays come. A desk lamp is an example of a positional light source. You can see the difference between directional and positional lights in "Plate 12" in Appendix I. The light used in Example 5-1 is a directional one:

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

As shown, you supply a vector of four values (x, y, z, w) for the GL_POSITION parameter. If the last value, w, is zero, the corresponding light source is a directional one, and the (x, y, z) values describe its direction. This direction is transformed by the modelview matrix. By default,

GL_POSITION is (0, 0, 1, 0), which defines a directional light that points along the negative *z*-axis. (Note that nothing prevents you from creating a directional light with the direction of (0, 0, 0), but such a light won't help you much.)

If the *w* value is nonzero, the light is positional, and the (x, y, z) values specify the location of the light in homogeneous object coordinates. (See Appendix F.) This location is transformed by the modelview matrix and stored in eye coordinates. (See "Controlling a Light's Position and Direction" for more information about how to control the transformation of the light's location.) Also, by default, a positional light radiates in all directions, but you can restrict it to producing a cone of illumination by defining the light as a spotlight. (See "Spotlights" for an explanation of how to define a light as a spotlight.)

Note: Remember that the colors across the face of a smooth-shaded polygon are determined by the colors calculated for the vertices. Because of this, you probably want to avoid using large polygons with local lights. If you locate the light near the middle of the polygon, the vertices might be too far away to receive much light, and the whole polygon will look darker than you intended. To avoid this problem, break up the large polygon into smaller ones.

For real-world lights, the intensity of light decreases as distance from the light increases. Since a directional light is infinitely far away, it doesn't make sense to attenuate its intensity over distance, so attenuation is disabled for a directional light. However, you might want to attenuate the light from a positional light. OpenGL attenuates a light source by multiplying the contribution of that source by an attenuation factor:

attenuation factor =
$$\frac{1}{k_c + k_l d + k_g d^2}$$

where

d = distance between the light's position and the vertex

kc = GL_CONSTANT_ATTENUATION

kl = GL_LINEAR_ATTENUATION

*k*q = GL_QUADRATIC_ATTENUATION

By default, *kc* is 1.0 and both *k*l and *kq* are zero, but you can give these parameters different values:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Note that the ambient, diffuse, and specular contributions are all attenuated. Only the emission and global ambient values aren't attenuated. Also note that since attenuation requires an additional division (and possibly more math) for each calculated color, using attenuated lights may slow down application performance.

Spotlights

As previously mentioned, you can have a positional light source act as a spotlight - that is, by

restricting the shape of the light it emits to a cone. To create a spotlight, you need to determine the spread of the cone of light you desire. (Remember that since spotlights are positional lights, you also have to locate them where you want them. Again, note that nothing prevents you from creating a directional spotlight, but it won't give you the result you want.) To specify the angle between the axis of the cone and a ray along the edge of the cone, use the GL_SPOT_CUTOFF parameter. The angle of the cone at the apex is then twice this value, as shown in Figure 5-2.

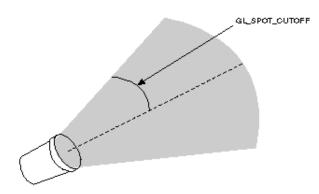


Figure 5-2 : GL_SPOT_CUTOFF Parameter

Note that no light is emitted beyond the edges of the cone. By default, the spotlight feature is disabled because the GL_SPOT_CUTOFF parameter is 180.0. This value means that light is emitted in all directions (the angle at the cone's apex is 360 degrees, so it isn't a cone at all). The value for GL_SPOT_CUTOFF is restricted to being within the range [0.0,90.0] (unless it has the special value 180.0). The following line sets the cutoff parameter to 45 degrees:

glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);

You also need to specify a spotlight's direction, which determines the axis of the cone of light:

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

The direction is specified in object coordinates. By default, the direction is (0.0, 0.0, -1.0), so if you don't explicitly set the value of GL_SPOT_DIRECTION, the light points down the negative *z*-axis. Also, keep in mind that a spotlight's direction is transformed by the modelview matrix just as though it were a normal vector, and the result is stored in eye coordinates. (See "Controlling a Light's Position and Direction" for more information about such transformations.)

In addition to the spotlight's cutoff angle and direction, there are two ways you can control the intensity distribution of the light within the cone. First, you can set the attenuation factor described earlier, which is multiplied by the light's intensity. You can also set the GL_SPOT_EXPONENT parameter, which by default is zero, to control how concentrated the light is. The light's intensity is highest in the center of the cone. It's attenuated toward the edges of the cone by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lit, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source. (See "The Mathematics of Lighting" for more details on the equations used to calculate light intensity.)

Multiple Lights

As mentioned, you can have at least eight lights in your scene (possibly more, depending on your OpenGL implementation). Since OpenGL needs to perform calculations to determine how much light each vertex receives from each light source, increasing the number of lights adversely affects performance. The constants used to refer to the eight lights are GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, GL_LIGHT3, and so on. In the preceding discussions, parameters related to GL_LIGHT0 were set. If you want an additional light, you need to specify its parameters; also, remember that the default values are different for these other lights than they are for GL_LIGHT0, as explained in Table 5-1. Example 5-3 defines a white attenuated spotlight.

Example 5-3 : Second Light Source

```
GLfloat light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_position[] = { -2.0, 2.0, 1.0, 1.0 };
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightf(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);
```

```
glEnable(GL_LIGHT1);
```

If these lines were added to Example 5-1, the sphere would be lit with two lights, one directional and one spotlight.

Try This

Modify Example 5-1 in the following manner:

- Change the first light to be a positional colored light rather than a directional white one.
- Add an additional colored spotlight. Hint: Use some of the code shown in the preceding section.
- Measure how these two changes affect performance.

Controlling a Light's Position and Direction

OpenGL treats the position and direction of a light source just as it treats the position of a geometric primitive. In other words, a light source is subject to the same matrix transformations as a primitive. More specifically, when **glLight***() is called to specify the position or the direction of a light source, the position or direction is transformed by the current modelview matrix and stored in eye coordinates. This means you can manipulate a light source's position or direction by changing the contents of the modelview matrix. (The projection matrix has no effect on a light's position or direction.) This section explains how to achieve the following three different effects by changing the point in the program at which the light position is set, relative to modeling or viewing

transformations:

- A light position that remains fixed
- A light that moves around a stationary object
- A light that moves along with the viewpoint

Keeping the Light Stationary

In the simplest example, as in Example 5-1, the light position remains fixed. To achieve this effect, you need to set the light position after whatever viewing and/or modeling transformation you use. In Example 5-4, the relevant code from the **init()** and **reshape()** routines might look like this.

Example 5-4 : Stationary Light Source

```
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
if (w <= h)
    glOrtho (-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
    glOrtho (-1.5*w/h, 1.5*w/h, -1.5, 1.5, -10.0, 10.0);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity();
/* later in init() */
GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

As you can see, the viewport and projection matrices are established first. Then, the identity matrix is loaded as the modelview matrix, after which the light position is set. Since the identity matrix is used, the originally specified light position (1.0, 1.0, 1.0) isn't changed by being multiplied by the modelview matrix. Then, since neither the light position nor the modelview matrix is modified after this point, the direction of the light remains (1.0, 1.0, 1.0).

Independently Moving the Light

Now suppose you want to rotate or translate the light position so that the light moves relative to a stationary object. One way to do this is to set the light position after the modeling transformation, which is itself changed specifically to modify the light position. You can begin with the same series of calls in **init**() early in the program. Then you need to perform the desired modeling transformation (on the modelview stack) and reset the light position, probably in **display**(). Example 5-5 shows what **display**() might be.

Example 5-5 : Independently Moving Light Source

```
static GLdouble spin;
void display(void)
{
    GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glPushMatrix();
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

```
glPushMatrix();
    glRotated(spin, 1.0, 0.0, 0.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glPopMatrix();
    glutSolidTorus (0.275, 0.85, 8, 15);
    glPopMatrix();
    glFlush();
}
```

spin is a global variable and is probably controlled by an input device. **display()** causes the scene to be redrawn with the light rotated *spin* degrees around a stationary torus. Note the two pairs of **glPushMatrix()** and **glPopMatrix()** calls, which are used to isolate the viewing and modeling transformations, all of which occur on the modelview stack. Since in Example 5-5 the viewpoint remains constant, the current matrix is pushed down the stack and then the desired viewing transformation is loaded with **gluLookAt()**. The matrix stack is pushed again before the modeling transformation **glRotated()** is specified. Then the light position is set in the new, rotated coordinate system so that the light itself appears to be rotated from its previous position. (Remember that the light position is stored in eye coordinates, which are obtained after transformation by the modelview matrix.) After the rotated matrix is popped off the stack, the torus is drawn.

Example 5-6 is a program that rotates a light source around an object. When the left mouse button is pressed, the light position rotates an additional 30 degrees. A small, unlit, wireframe cube is drawn to represent the position of the light in the scene.

Example 5-6 : Moving a Light with Modeling Transformations: movelight.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
static int spin = 0;
void init(void)
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL SMOOTH);
  glEnable(GL LIGHTING);
  glEnable(GL LIGHT0);
  glEnable(GL_DEPTH_TEST);
}
/* Here is where the light position is reset after the modeling
  transformation (glRotated) is called. This places the
 * light at a new position in world coordinates. The cube
 *
   represents the position of the light.
 * /
void display(void)
  GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();
   glTranslatef (0.0, 0.0, -5.0);
   glPushMatrix ();
   glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
   glLightfv (GL_LIGHT0, GL_POSITION, position);
   glTranslated (0.0, 0.0, 1.5);
   glDisable (GL_LIGHTING);
```

```
glColor3f (0.0, 1.0, 1.0);
   glutWireCube (0.1);
   glEnable (GL_LIGHTING);
   glPopMatrix ();
   glutSolidTorus (0.275, 0.85, 8, 15);
   glPopMatrix ();
  glFlush ();
}
void reshape (int w, int h)
ł
  glViewport (0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL PROJECTION);
  glLoadIdentity();
  gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
   switch (button) {
      case GLUT_LEFT BUTTON:
         if (state == GLUT DOWN) {
            spin = (spin + 30) % 360;
            glutPostRedisplay();
         break;
      default:
         break;
   }
ļ
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT SINGLE | GLUT RGB | GLUT DEPTH);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
  glutMainLoop();
  return 0;
}
```

Moving the Light Source Together with Your Viewpoint

To create a light that moves along with the viewpoint, you need to set the light position before the viewing transformation. Then the viewing transformation affects both the light and the viewpoint in the same way. Remember that the light position is stored in eye coordinates, and this is one of the few times when eye coordinates are critical. In Example 5-7, the light position is defined in **init**(), which stores the light position at (0, 0, 0) in eye coordinates. In other words, the light is shining from the lens of the camera.

Example 5-7 : Light Source That Moves with the Viewpoint

```
GLfloat light_position() = { 0.0, 0.0, 0.0, 1.0 };
glViewport(0, 0, (GLint) w, (GLint) h);
```

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

If the viewpoint is now moved, the light will move along with it, maintaining (0, 0, 0) distance, relative to the eye. In the continuation of Example 5-7, which follows next, the global variables (*ex*, *ey*, *ez*) and (*upx*, *upy*, *upz*) control the position of the viewpoint and up vector. The **display**() routine that's called from the event loop to redraw the scene might be this:

```
static GLdouble ex, ey, ez, upx, upy, upz;
void display(void)
{
    glClear(GL_COLOR_BUFFER_MASK | GL_DEPTH_BUFFER_MASK);
    glPushMatrix();
    gluLookAt (ex, ey, ez, 0.0, 0.0, 0.0, upx, upy, upz);
    glutSolidTorus (0.275, 0.85, 8, 15);
    glPopMatrix();
    glFlush();
}
```

When the lit torus is redrawn, both the light position and the viewpoint are moved to the same location. As the values passed to **gluLookAt()** change and the eye moves, the object will never appear dark, because it is always being illuminated from the eye position. Even though you haven't respecified the light position, the light moves because the eye coordinate system has changed.

This method of moving the light can be very useful for simulating the illumination from a miner's hat. Another example would be carrying a candle or lantern. The light position specified by the call to **glLightfv**(GL_LIGHTi, GL_POSITION, position) would be the x, y, and z distance from the eye position to the illumination source. Then as the eye position moves, the light will remain the same relative distance away.

Try This

Modify Example 5-6 in the following manner:

- Make the light translate past the object instead of rotating around it. Hint: Use **glTranslated**() rather than the first **glRotated**() in **display**(), and choose an appropriate value to use instead of *spin*.
- Change the attenuation so that the light decreases in intensity as it's moved away from the object. Hint: Add calls to **glLight***() to set the desired attenuation parameters.

Selecting a Lighting Model

The OpenGL notion of a lighting model has three components:

• The global ambient light intensity

- Whether the viewpoint position is local to the scene or whether it should be considered to be an infinite distance away
- Whether lighting calculations should be performed differently for both the front and back faces of objects

This section explains how to specify a lighting model. It also discusses how to enable lighting - that is, how to tell OpenGL that you want lighting calculations performed.

The command used to specify all properties of the lighting model is **glLightModel***(). **glLightModel***() has two arguments: the lighting model property and the desired value for that property.

void glLightModel{if}(GLenum pname, TYPEparam);

void glLightModel{if}v(GLenum pname, TYPE *param);

Sets properties of the lighting model. The characteristic of the lighting model being set is defined by pname, which specifies a named parameter (see Table 5-2). param indicates the values to which the pname characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued lighting model characteristics, not for GL_LIGHT_MODEL_AMBIENT.

Parameter Name	Default Value	Meaning
GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient RGBA intensity of the entire scene
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0 or GL_FALSE	how specular reflection angles are computed
GL_LIGHT_MODEL_TWO_SIDE	0.0 or GL_FALSE	choose between one-sided or two-sided lighting

 Table 5-2 : Default Values for pname Parameter of glLightModel*()

Global Ambient Light

As discussed earlier, each light source can contribute ambient light to a scene. In addition, there can be other ambient light that's not from any particular source. To specify the RGBA intensity of such global ambient light, use the GL_LIGHT_MODEL_AMBIENT parameter as follows:

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

In this example, the values used for *lmodel_ambient* are the default values for GL_LIGHT_MODEL_AMBIENT. Since these numbers yield a small amount of white ambient light, even if you don't add a specific light source to your scene, you can still see the objects in the scene. "Plate 14" in Appendix I shows the effect of different amounts of global ambient light.

Local or Infinite Viewpoint

The location of the viewpoint affects the calculations for highlights produced by specular reflectance. More specifically, the intensity of the highlight at a particular vertex depends on the normal at that vertex, the direction from the vertex to the light source, and the direction from the vertex to the viewpoint. Keep in mind that the viewpoint isn't actually being moved by calls to lighting commands (you need to change the projection transformation, as described in "Projection Transformations" in Chapter 3); instead, different assumptions are made for the lighting calculations as if the viewpoint were moved.

With an infinite viewpoint, the direction between it and any vertex in the scene remains constant. A local viewpoint tends to yield more realistic results, but since the direction has to be calculated for each vertex, overall performance is decreased with a local viewpoint. By default, an infinite viewpoint is assumed. Here's how to change to a local viewpoint:

glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

This call places the viewpoint at (0, 0, 0) in eye coordinates. To switch back to an infinite viewpoint, pass in GL_FALSE as the argument.

Two-sided Lighting

Lighting calculations are performed for all polygons, whether they're front-facing or back-facing. Since you usually set up lighting conditions with the front-facing polygons in mind, however, the back-facing ones typically aren't correctly illuminated. In Example 5-1 where the object is a sphere, only the front faces are ever seen, since they're the ones on the outside of the sphere. So, in this case, it doesn't matter what the back-facing polygons look like. If the sphere is going to be cut away so that its inside surface will be visible, however, you might want to have the inside surface be fully lit according to the lighting conditions you've defined; you might also want to supply a different material description for the back faces. When you turn on two-sided lighting with

glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

OpenGL reverses the surface normals for back-facing polygons; typically, this means that the surface normals of visible back- and front-facing polygons face the viewer, rather than pointing away. As a result, all polygons are illuminated correctly. However, these additional operations usually make two-sided lighting perform more slowly than the default one-sided lighting.

To turn two-sided lighting off, pass in GL_FALSE as the argument in the preceding call. (See "Defining Material Properties" for information about how to supply material properties for both faces.) You can also control which faces OpenGL considers to be front-facing with the command **glFrontFace**(). (See "Reversing and Culling Polygon Faces" in Chapter 2 for more information.)

Enabling Lighting

With OpenGL, you need to explicitly enable (or disable) lighting. If lighting isn't enabled, the current color is simply mapped onto the current vertex, and no calculations concerning normals, light sources, the lighting model, and material properties are performed. Here's how to enable lighting:

```
glEnable(GL_LIGHTING);
```

To disable lighting, call **glDisable**() with GL_LIGHTING as the argument.

You also need to explicitly enable each light source that you define, after you've specified the parameters for that source. Example 5-1 uses only one light, GL_LIGHT0:

glEnable(GL_LIGHT0);

Defining Material Properties

You've seen how to create light sources with certain characteristics and how to define the desired lighting model. This section describes how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors, the shininess, and the color of any emitted light. (See "The Mathematics of Lighting" for the equations used in the lighting and material-property calculations.) Most of the material properties are conceptually similar to ones you've already used to create light sources. The mechanism for setting them is similar, except that the command used is called **glMaterial***().

void glMaterial{if}(GLenum face, GLenum pname, TYPEparam);

void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);

Specifies a current material property for use in lighting calculations. face can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate which face of the object the material should be applied to. The particular material property being set is identified by pname and the desired values for that property are given by param, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector version is used). The nonvector version works only for setting GL_SHININESS. The possible values for pname are shown in Table 5-3. Note that GL_AMBIENT_AND_DIFFUSE allows you to set both the ambient and diffuse material colors simultaneously to the same RGBA value.

Table 5-3 : Default Values for pname Parameter of glMaterial*()

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient color of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular color of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive color of material
GL_COLOR_INDEXES	(0,1,1)	ambient, diffuse, and specular color indices

As discussed in "Selecting a Lighting Model," you can choose to have lighting calculations performed differently for the front- and back-facing polygons of objects. If the back faces might indeed be seen, you can supply different material properties for the front and the back surfaces by using the *face* parameter of **glMaterial***(). See "Plate 14" in Appendix I for an example of an object drawn with different inside and outside material properties.

To give you an idea of the possible effects you can achieve by manipulating material properties, see "Plate 16" in Appendix I. This figure shows the same object drawn with several different sets of material properties. The same light source and lighting model are used for the entire figure. The sections that follow discuss the specific properties used to draw each of these spheres.

Note that most of the material properties set with **glMaterial***() are (R, G, B, A) colors. Regardless of what alpha values are supplied for other parameters, the alpha value at any particular vertex is the diffuse-material alpha value (that is, the alpha value given to GL_DIFFUSE with the **glMaterial***() command, as described in the next section). (See "Blending" in Chapter 6 for a complete discussion of alpha values.) Also, none of the RGBA material properties apply in color-index mode. (See "Lighting in Color-Index Mode" for more information about what parameters are relevant in color-index mode.)

Diffuse and Ambient Reflection

The GL_DIFFUSE and GL_AMBIENT parameters set with **glMaterial***() affect the color of the diffuse and ambient light reflected by an object. Diffuse reflectance plays the most important role in determining what you perceive the color of an object to be. It's affected by the color of the incident diffuse light and the angle of the incident light relative to the normal direction. (It's most intense where the incident light falls perpendicular to the surface.) The position of the viewpoint doesn't affect diffuse reflectance at all.

Ambient reflectance affects the overall color of the object. Because diffuse reflectance is brightest

where an object is directly illuminated, ambient reflectance is most noticeable where an object receives no direct illumination. An object's total ambient reflectance is affected by the global ambient light and ambient light from individual light sources. Like diffuse reflectance, ambient reflectance isn't affected by the position of the viewpoint.

For real-world objects, diffuse and ambient reflectance are normally the same color. For this reason, OpenGL provides you with a convenient way of assigning the same value to both simultaneously with **glMaterial***():

In this example, the RGBA color (0.1, 0.5, 0.8, 1.0) - a deep blue color - represents the current ambient and diffuse reflectance for both the front- and back-facing polygons.

In "Plate 16" in Appendix I, the first row of spheres has no ambient reflectance (0.0, 0.0, 0.0, 0.0), and the second row has a significant amount of it (0.7, 0.7, 0.7, 1.0).

Specular Reflection

Specular reflection from an object produces highlights. Unlike ambient and diffuse reflection, the amount of specular reflection seen by a viewer does depend on the location of the viewpoint - it's brightest along the direct angle of reflection. To see this, imagine looking at a metallic ball outdoors in the sunlight. As you move your head, the highlight created by the sunlight moves with you to some extent. However, if you move your head too much, you lose the highlight entirely.

OpenGL allows you to set the effect that the material has on reflected light (with GL_SPECULAR) and control the size and brightness of the highlight (with GL_SHININESS). You can assign a number in the range of [0.0, 128.0] to GL_SHININESS - the higher the value, the smaller and brighter (more focused) the highlight. (See "The Mathematics of Lighting" for the details of how specular highlights are calculated.)

In "Plate 16" in Appendix I, the spheres in the first column have no specular reflection. In the second column, GL_SPECULAR and GL_SHININESS are assigned values as follows:

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
```

In the third column, the GL_SHININESS parameter is increased to 100.0.

Emission

By specifying an RGBA color for GL_EMISSION, you can make an object appear to be giving off light of that color. Since most real-world objects (except lights) don't emit light, you'll probably use this feature mostly to simulate lamps and other light sources in a scene. In "Plate 16" in Appendix I, the spheres in the fourth column have a reddish, grey value for GL_EMISSION:

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

Notice that the spheres appear to be slightly glowing; however, they're not actually acting as light sources. You would need to create a light source and position it at the same location as the sphere to create that effect.

Changing Material Properties

Example 5-1 uses the same material properties for all vertices of the only object in the scene (the sphere). In other situations, you might want to assign different material properties for different vertices on the same object. More likely, you have more than one object in the scene, and each object has different material properties. For example, the code that produced "Plate 16" in Appendix I has to draw twelve different objects (all spheres), each with different material properties. Example 5-8 shows a portion of the code in **display**().

Example 5-8 : Different Material Properties: material.c

```
GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
   GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 };
   GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat no_shininess[] = { 0.0 };
   GLfloat low_shininess[] = { 5.0 };
   GLfloat high_shininess[] = { 100.0 };
   GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
   glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
/* draw sphere in first row, first column
  diffuse reflection only; no ambient or specular
 * /
   glPushMatrix();
   glTranslatef (-3.75, 3.0, 0.0);
   glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
   glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
   glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
   glutSolidSphere(1.0, 16, 16);
   glPopMatrix();
/* draw sphere in first row, second column
 * diffuse and specular reflection; low shininess; no ambient
 * /
   glPushMatrix();
   glTranslatef (-1.25, 3.0, 0.0);
  glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
   glutSolidSphere(1.0, 16, 16);
   glPopMatrix();
/* draw sphere in first row, third column
  diffuse and specular reflection; high shininess; no ambient
 * /
   glPushMatrix();
   glTranslatef (1.25, 3.0, 0.0);
   glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL FRONT, GL SPECULAR, mat specular);
```

```
glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
  glMaterialfv(GL FRONT, GL EMISSION, no mat);
  glutSolidSphere(1.0, 16, 16);
  glPopMatrix();
/* draw sphere in first row, fourth column
 * diffuse reflection; emission; no ambient or specular refl.
 * /
  glPushMatrix();
  glTranslatef (3.75, 3.0, 0.0);
  glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
  glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
  glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
  glutSolidSphere(1.0, 16, 16);
  glPopMatrix();
```

As you can see, **glMaterialfv()** is called repeatedly to set the desired material property for each sphere. Note that it only needs to be called to change a property that needs to be respecified. The second, third, and fourth spheres use the same ambient and diffuse properties as the first sphere, so these properties do not need to be respecified. Since **glMaterial***() has a performance cost associated with its use, Example 5-8 could be rewritten to minimize material-property changes.

Another technique for minimizing performance costs associated with changing material properties is to use **glColorMaterial**().

void glColorMaterial(GLenum face, GLenum mode);

Causes the material property (or properties) specified by mode of the specified material face (or faces) specified by face to track the value of the current color at all times. A change to the current color (using **glColor***()) immediately updates the specified material properties. The face parameter can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK (the default). The mode parameter can be GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE (the default), GL_SPECULAR, or GL_EMISSION. At any given time, only one mode is active. **glColorMaterial**() has no effect on color-index lighting.

Note that **glColorMaterial**() specifies two independent values: the first specifies which face or faces are updated, and the second specifies which material property or properties of those faces are updated. OpenGL does *not* maintain separate *mode* variables for each face.

After calling **glColorMaterial**(), you need to call **glEnable**() with GL_COLOR_MATERIAL as the parameter. Then, you can change the current color using **glColor***() (or other material properties, using **glMaterial***()) as needed as you draw:

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
/* now glColor* changes diffuse reflection */
glColor3f(0.2, 0.5, 0.8);
/* draw some objects here */
glColorMaterial(GL_FRONT, GL_SPECULAR);
/* glColor* no longer changes diffuse reflection */
/* now glColor* changes specular reflection */
glColor3f(0.9, 0.0, 0.2);
/* draw other objects here */
glDisable(GL_COLOR_MATERIAL);
```

You should use glColorMaterial() whenever you need to change a single material parameter for

most vertices in your scene. If you need to change more than one material parameter, as was the case for "Plate 16" in Appendix I, use **glMaterial***(). When you don't need the capabilities of **glColorMaterial**() anymore, be sure to disable it so that you don't get undesired material properties and don't incur the performance cost associated with it. The performance value in using **glColorMaterial**() varies, depending on your OpenGL implementation. Some implementations may be able to optimize the vertex routines so that they can quickly update material properties based on the current color.

Example 5-9 shows an interactive program that uses **glColorMaterial**() to change material parameters. Pressing each of the three mouse buttons changes the color of the diffuse reflection.

Example 5-9 : Using glColorMaterial(): colormat.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
GLfloat diffuseMaterial[4] = { 0.5, 0.5, 0.5, 1.0 };
void init(void)
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL SMOOTH);
   glEnable(GL DEPTH TEST);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glColorMaterial(GL_FRONT, GL_DIFFUSE);
   glEnable(GL COLOR MATERIAL);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glutSolidSphere(1.0, 20, 16);
   glFlush ();
}
void reshape (int w, int h)
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL PROJECTION);
   glLoadIdentity();
   if (w \le h)
      glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
         1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
         1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
   glMatrixMode(GL MODELVIEW);
   glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
```

```
{
   switch (button) {
      case GLUT LEFT BUTTON:
                                              change red */
         if (state == GLUT DOWN) {
                                           /*
            diffuseMaterial[0] += 0.1;
            if (diffuseMaterial[0] > 1.0)
               diffuseMaterial[0] = 0.0;
            glColor4fv(diffuseMaterial);
            glutPostRedisplay();
         }
         break;
      case GLUT_MIDDLE_BUTTON:
         if (state == GLUT_DOWN) {
                                          /*
                                              change green */
            diffuseMaterial[1] += 0.1;
            if (diffuseMaterial[1] > 1.0)
               diffuseMaterial[1] = 0.0;
            glColor4fv(diffuseMaterial);
            glutPostRedisplay();
         }
         break;
      case GLUT RIGHT BUTTON:
         if (state == GLUT_DOWN) {
                                         /*
                                             change blue */
            diffuseMaterial[2] += 0.1;
            if (diffuseMaterial[2] > 1.0)
               diffuseMaterial[2] = 0.0;
            glColor4fv(diffuseMaterial);
            glutPostRedisplay();
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
ł
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

Try This

Modify Example 5-8 in the following manner:

- Change the global ambient light in the scene. Hint: Alter the value of the GL_LIGHT_MODEL_AMBIENT parameter.
- Change the diffuse, ambient, and specular reflection parameters, the shininess exponent, and the emission color. Hint: Use the **glMaterial***() command, but avoid making excessive calls.
- Use two-sided materials and add a user-defined clipping plane so that you can see the inside and outside of a row or column of spheres. (See "Additional Clipping Planes" in Chapter 3, if you need to recall user-defined clipping planes.) Hint: Turn on two-sided lighting with

GL_LIGHT_MODEL_TWO_SIDE, set the desired material properties, and add a clipping plane.

• Remove all the **glMaterialfv**() calls, and use the more efficient **glColorMaterial**() calls to achieve the same lighting.

The Mathematics of Lighting

Advanced

This section presents the equations used by OpenGL to perform lighting calculations to determine colors when in RGBA mode. (See "The Mathematics of Color-Index Mode Lighting" for corresponding calculations for color-index mode.) You don't need to read this section if you're willing to experiment to obtain the lighting conditions you want. Even after reading this section, you'll probably have to experiment, but you'll have a better idea of how the values of parameters affect a vertex's color. Remember that if lighting is not enabled, the color of a vertex is simply the current color; if it is enabled, the lighting computations described here are carried out in eye coordinates.

In the following equations, mathematical operations are performed separately on the R, G, and B components. Thus, for example, when three terms are shown as added together, the R values, the G values, and the B values for each term are separately added to form the final RGB color (R1+R2+R3, G1+G2+G3, B1+B2+B3). When three terms are multiplied, the calculation is (R1R2R3, G1G2G3, B1B2B3). (Remember that the final A or alpha component at a vertex is equal to the material's diffuse alpha value at that vertex.)

The color produced by lighting a vertex is computed as follows:

vertex color =

the material emission at that vertex +

the global ambient light scaled by the material's ambient property at that vertex +

the ambient, diffuse, and specular contributions from all the light sources, properly attenuated

After lighting calculations are performed, the color values are clamped (in RGBA mode) to the range [0,1].

Note that OpenGL lighting calculations don't take into account the possibility of one object blocking light from another; as a result shadows aren't automatically created. (See "Shadows" in Chapter 14 for a technique to create shadows.) Also keep in mind that with OpenGL, illuminated objects don't radiate light onto other objects.

Material Emission

The material emission term is the simplest. It's the RGB value assigned to the GL_EMISSION parameter.

Scaled Global Ambient Light

The second term is computed by multiplying the global ambient light (as defined by the GL_LIGHT_MODEL_AMBIENT parameter) by the material's ambient property (GL_AMBIENT value as assigned with **glMaterial***()):

ambientlight model * ambientmaterial

Each of the R, G, and B values for these two parameters are multiplied separately to compute the final RGB value for this term: (R1R2, G1G2, B1B2).

Contributions from Light Sources

Each light source may contribute to a vertex's color, and these contributions are added together. The equation for computing each light source's contribution is as follows:

contribution = attenuation factor * spotlight effect *

(ambient term + diffuse term + specular term)

Attenuation Factor

The attenuation factor was described in "Position and Attenuation":

attenuation factor = $\frac{1}{k_c + k_l d + k_q d^2}$

where

d = distance between the light's position and the vertex

kc = GL_CONSTANT_ATTENUATION

kl = GL_LINEAR_ATTENUATION

kq = GL_QUADRATIC_ATTENUATION

If the light is a directional one, the attenuation factor is 1.

Spotlight Effect

The *spotlight effect* evaluates to one of three possible values, depending on whether the light is actually a spotlight and whether the vertex lies inside or outside the cone of illumination produced by the spotlight:

- 1 if the light isn't a spotlight (GL_SPOT_CUTOFF is 180.0).
- 0 if the light is a spotlight, but the vertex lies outside the cone of illumination produced by the spotlight.

• $(\max \{\mathbf{v} \cdot \mathbf{d}, 0\})$ GL_SPOT_EXPONENT where:

 $\mathbf{v} = (vx, vy, vz)$ is the unit vector that points from the spotlight (GL_POSITION) to the vertex.

 $\mathbf{d} = (dx, dy, dz)$ is the spotlight's direction (GL_SPOT_DIRECTION), assuming the light is a spotlight and the vertex lies inside the cone of illumination produced by the spotlight.

The dot product of the two vectors \mathbf{v} and \mathbf{d} varies as the cosine of the angle between them; hence, objects directly in line get maximum illumination, and objects off the axis have their illumination drop as the cosine of the angle.

To determine whether a particular vertex lies within the cone of illumination, OpenGL evaluates (max $\{\mathbf{v} \cdot \mathbf{d}, 0\}$) where \mathbf{v} and \mathbf{d} are as defined in the preceding discussion. If this value is less than the cosine of the spotlight's cutoff angle (GL_SPOT_CUTOFF), then the vertex lies outside the cone; otherwise, it's inside the cone.

Ambient Term

The ambient term is simply the ambient color of the light scaled by the ambient material property:

ambientlight *ambientmaterial

Diffuse Term

The diffuse term needs to take into account whether light falls directly on the vertex, the diffuse color of the light, and the diffuse material property:

 $(\max \{\mathbf{L} \cdot \mathbf{n}, 0\}) * \text{diffuselight } * \text{diffusematerial}$

where:

L = (Lx, Ly, Lz) is the unit vector that points from the vertex to the light position (GL_POSITION).

 $\mathbf{n} = (\mathbf{n}x, \mathbf{n}y, \mathbf{n}z)$ is the unit normal vector at the vertex.

Specular Term

The specular term also depends on whether light falls directly on the vertex. If $\mathbf{L} \cdot \mathbf{n}$ is less than or equal to zero, there is no specular component at the vertex. (If it's less than zero, the light is on the wrong side of the surface.) If there's a specular component, it depends on the following:

- The unit normal vector at the vertex (nx, ny, nz).
- The sum of the two unit vectors that point between (1) the vertex and the light position (or light direction) and (2) the vertex and the viewpoint (assuming that GL_LIGHT_MODEL_LOCAL_VIEWER is true; if it's not true, the vector (0, 0, 1) is used as the second vector in the sum). This vector sum is normalized (by dividing each component by the magnitude of the vector) to yield **s** = (*s*x, *s*y, *s*z).
- The specular exponent (GL_SHININESS).

- The specular color of the light (GL_SPECULARlight).
- The specular property of the material (GL_SPECULARmaterial).

Using these definitions, here's how OpenGL calculates the specular term:

 $(\max \{\mathbf{s} \cdot \mathbf{n}, 0\})$ shininess * specularlight * specularmaterial

However, if $\mathbf{L} \cdot \mathbf{n} = 0$, the specular term is 0.

Putting It All Together

Using the definitions of terms described in the preceding paragraphs, the following represents the entire lighting calculation in RGBA mode:

vertex color = emissionmaterial +

ambientlight model * ambientmaterial +

$$\sum_{i=0}^{n-1} \left(\frac{1}{k_c + k_i d + k_q d^2} \right)_i^* (\text{spotlight effect})_i^*$$

[ambientlight *ambientmaterial +

 $(\max \{ \mathbf{L} \cdot \mathbf{n}, 0 \}) * diffuselight * diffusematerial +$

(max { $\mathbf{s} \cdot \mathbf{n}$, 0}) shininess * specularlight * specularmaterial] i

Lighting in Color-Index Mode

In color-index mode, the parameters comprising RGBA values either have no effect or have a special interpretation. Since it's much harder to achieve certain effects in color-index mode, you should use RGBA whenever possible. In fact, the only light-source, lighting-model, or material parameters in an RGBA form that are used in color-index mode are the light-source parameters GL_DIFFUSE and GL_SPECULAR and the material parameter GL_SHININESS. GL_DIFFUSE and GL_SPECULAR (*d*l and *s*l, respectively) are used to compute color-index diffuse and specular light intensities (*d*ci and *s*ci) as follows:

dci = 0.30 R(dl) + 0.59 G(dl) + 0.11 B(dl)

sci = 0.30 R(sl) + 0.59 G(sl) + 0.11 B(sl)

where R(x), G(x), and B(x) refer to the red, green, and blue components, respectively, of color x. The weighting values 0.30, 0.59, and 0.11 reflect the "perceptual" weights that red, green, and blue have for your eye - your eye is most sensitive to green and least sensitive to blue.

To specify material colors in color-index mode, use glMaterial*() with the special parameter

GL_COLOR_INDEXES, as follows:

GLfloat mat_colormap[] = { 16.0, 47.0, 79.0 };
glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);

The three numbers supplied for GL_COLOR_INDEXES specify the color indices for the ambient, diffuse, and specular material colors, respectively. In other words, OpenGL regards the color associated with the first index (16.0 in this example) as the pure ambient color, with the second index (47.0) as the pure diffuse color, and with the third index (79.0) as the pure specular color. (By default, the ambient color index is 0.0, and the diffuse and specular color indices are both 1.0. Note that **glColorMaterial**() has no effect on color-index lighting.)

As it draws a scene, OpenGL uses colors associated with indices in between these numbers to shade objects in the scene. Therefore, you must build a color ramp between the indicated indices (in this example, between indices 16 and 47, and then between 47 and 79). Often, the color ramp is built smoothly, but you might want to use other formulations to achieve different effects. Here's an example of a smooth color ramp that starts with a black ambient color and goes through a magenta diffuse color to a white specular color:

```
for (i = 0; i < 32; i++) {
  glutSetColor (16 + i, 1.0 * (i/32.0), 0.0, 1.0 * (i/32.0));
  glutSetColor (48 + i, 1.0, 1.0 * (i/32.0), 1.0);
}</pre>
```

The GLUT library command **glutSetColor**() takes four arguments. It associates the color index indicated by the first argument to the RGB triplet specified by the last three arguments. When i = 0, the color index 16 is assigned the RGB value (0.0, 0.0, 0.0), or black. The color ramp builds smoothly up to the diffuse material color at index 47 (when i = 31), which is assigned the pure magenta RGB value (1.0, 0.0, 1.0). The second loop builds the ramp between the magenta diffuse color and the white (1.0, 1.0, 1.0) specular color (index 79). "Plate 15" in Appendix I shows the result of using this color ramp with a single lit sphere.

The Mathematics of Color-Index Mode Lighting

Advanced

As you might expect, since the allowable parameters are different for color-index mode than for RGBA mode, the calculations are different as well. Since there's no material emission and no ambient light, the only terms of interest from the RGBA equations are the diffuse and specular contributions from the light sources and the shininess. Even these need to be modified, however, as explained next.

Begin with the diffuse and specular terms from the RGBA equations. In the diffuse term, instead of diffuselight * diffusematerial, substitute *d*ci as defined in the previous section for color-index mode. Similarly, in the specular term, instead of specularlight * specularmaterial, use *s*ci as defined in the previous section. (Calculate the attenuation, spotlight effect, and all other components of these terms as before.) Call these modified diffuse and specular terms *d* and *s*, respectively. Now let $s' = \min\{s, 1\}$, and then compute

 $c = a\mathbf{m} + d(1-s')(d\mathbf{m}-a\mathbf{m}) + s'(s\mathbf{m}-a\mathbf{m})$

where *a*m, *d*m, and *s*m are the ambient, diffuse, and specular material indexes specified using GL_COLOR_INDEXES. The final color index is

 $c' = \min \{ c, sm \}$

After lighting calculations are performed, the color-index values are converted to fixed-point (with an unspecified number of bits to the right of the binary point). Then the integer portion is masked (bitwise ANDed) with 2n-1, where *n* is the number of bits in a color in the color-index buffer.

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 6 Blending, Antialiasing, Fog, and Polygon Offset

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Blend colors to achieve such effects as making objects appear translucent
- Smooth jagged edges of lines and polygons with antialiasing
- Create scenes with realistic atmospheric effects
- Draw geometry at or near the same depth, but avoid unaesthetic artifacts from intersecting geometry

The preceding chapters have given you the basic information you need to create a computer-graphics scene; you've learned how to do the following:

- Draw geometric shapes
- Transform those geometric shapes so that they can be viewed from whatever perspective you wish
- Specify how the geometric shapes in your scene should be colored and shaded
- Add lights and indicate how they should affect the shapes in your scene

Now you're ready to get a little fancier. This chapter discusses four techniques that can add extra detail and polish to your scene. None of these techniques is hard to use - in fact, it's probably harder to explain them than to use them. Each of these techniques is described in its own major section:

- "Blending" tells you how to specify a blending function that combines color values from a source and a destination. The final effect is that parts of your scene appear translucent.
- "Antialiasing" explains this relatively subtle technique that alters colors so that the edges of points, lines, and polygons appear smooth rather than angular and jagged.
- "Fog" describes how to create the illusion of depth by computing the color values of an object based on its distance from the viewpoint. Thus, objects that are far away appear to fade into the background, just as they do in real life.

• If you've tried to draw a wireframe outline atop a shaded object and used the same vertices, you've probably noticed some ugly visual artifacts. "Polygon Offset" shows you how to tweak (offset) depth values to make an outlined, shaded object look beautiful.

Blending

You've already seen alpha values (alpha is the A in RGBA), but they've been ignored until now. Alpha values are specified with **glColor***(), when using **glClearColor**() to specify a clearing color and when specifying certain lighting parameters such as a material property or light-source intensity. As you learned in Chapter 4, the pixels on a monitor screen emit red, green, and blue light, which is controlled by the red, green, and blue color values. So how does an alpha value affect what gets drawn in a window on the screen?

When blending is enabled, the alpha value is often used to combine the color value of the fragment being processed with that of the pixel already stored in the framebuffer. Blending occurs after your scene has been rasterized and converted to fragments, but just before the final pixels are drawn in the framebuffer. Alpha values can also be used in the alpha test to accept or reject a fragment based on its alpha value. (See Chapter 10 for more information about this process.)

Without blending, each new fragment overwrites any existing color values in the framebuffer, as though the fragment were opaque. With blending, you can control how (and how much of) the existing color value should be combined with the new fragment's value. Thus you can use alpha blending to create a translucent fragment that lets some of the previously stored color value "show through." Color blending lies at the heart of techniques such as transparency, digital compositing, and painting.

Note: Alpha values aren't specified in color-index mode, so blending operations aren't performed in color-index mode.

The most natural way to think of blending operations is to think of the RGB components of a fragment as representing its color and the alpha component as representing opacity. Transparent or translucent surfaces have lower opacity than opaque ones and, therefore, lower alpha values. For example, if you're viewing an object through green glass, the color you see is partly green from the glass and partly the color of the object. The percentage varies depending on the transmission properties of the glass: If the glass transmits 80 percent of the light that strikes it (that is, has an opacity of 20 percent), the color you see is a combination of 20 percent glass color and 80 percent of the color of the object behind it. You can easily imagine situations with multiple translucent surfaces. If you look at an automobile, for instance, its interior has one piece of glass between it and your viewpoint; some objects behind the automobile are visible through two pieces of glass.

The Source and Destination Factors

During blending, color values of the incoming fragment (the *source*) are combined with the color values of the corresponding currently stored pixel (the *destination*) in a two-stage process. First you specify how to compute source and destination factors. These factors are RGBA quadruplets that are multiplied by each component of the R, G, B, and A values in the source and destination, respectively. Then the corresponding components in the two sets of RGBA quadruplets are added. To show this mathematically, let the source and destination blending factors be (Sr, Sg, Sb, Sa) and

(Dr, Dg, Db, Da), respectively, and the RGBA values of the source and destination be indicated with a subscript of s or d. Then the final, blended RGBA values are given by

(RsSr+RdDr, GsSg+GdDg, BsSb+BdDb, AsSa+AdDa)

Each component of this quadruplet is eventually clamped to [0,1].

Now consider how the source and destination blending factors are generated. You use **glBlendFunc()** to supply two constants: one that specifies how the source factor should be computed and one that indicates how the destination factor should be computed. To have blending take effect, you also need to enable it:

glEnable(GL_BLEND);

Use **glDisable**() with GL_BLEND to disable blending. Also note that using the constants GL_ONE (source) and GL_ZERO (destination) gives the same results as when blending is disabled; these values are the default.

void glBlendFunc(GLenum sfactor, GLenum dfactor);

Controls how color values in the fragment being processed (the source) are combined with those already stored in the framebuffer (the destination). The argument sfactor indicates how to compute a source blending factor; dfactor indicates how to compute a destination blending factor. The possible values for these arguments are explained in Table 6-1. The blend factors are assumed to lie in the range [0,1]; after the color values in the source and destination are combined, they're clamped to the range [0,1].

Note: In Table 6-1, the RGBA values of the source and destination are indicated with the subscripts s and d, respectively. Subtraction of quadruplets means subtracting them componentwise. The Relevant Factor column indicates whether the corresponding constant can be used to specify the source or destination blend factor.

Table 6-1 : Source and Destination Blending Factors

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(Rd, Gd, Bd, Ad)
GL_SRC_COLOR	destination	(Rs, Gs, Bs, As)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(Rd, Gd, Bd, Ad)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1)-(Rs, Gs, Bs, As)
GL_SRC_ALPHA	source or destination	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1, 1, 1, 1)-(As, As, As, As)
GL_DST_ALPHA	source or destination	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPHA	source or destination	(1, 1, 1, 1)-(Ad, Ad, Ad, Ad)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1-Ad)

Sample Uses of Blending

Not all combinations of source and destination factors make sense. Most applications use a small number of combinations. The following paragraphs describe typical uses for particular combinations of source and destination factors. Some of these examples use only the incoming alpha value, so they work even when alpha values aren't stored in the framebuffer. Also note that often there's more than one way to achieve some of these effects.

- One way to draw a picture composed half of one image and half of another, equally blended, is to set the source factor to GL_ONE and the destination factor to GL_ZERO, and draw the first image. Then set the source factor to GL_SRC_ALPHA and destination factor to GL_ONE_MINUS_SRC_ALPHA, and draw the second image with alpha equal to 0.5. This pair of factors probably represents the most commonly used blending operation. If the picture is supposed to be blended with 0.75 of the first image and 0.25 of the second, draw the first image as before, and draw the second with an alpha of 0.25.
- To blend three different images equally, set the destination factor to GL_ONE and the source factor to GL_SRC_ALPHA. Draw each of the images with an alpha equal to 0.3333333. With this technique, each image is only one-third of its original brightness, which is noticeable where the images don't overlap.
- Suppose you're writing a paint program, and you want to have a brush that gradually adds

color so that each brush stroke blends in a little more color with whatever is currently in the image (say 10 percent color with 90 percent image on each pass). To do this, draw the image of the brush with alpha of 10 percent and use GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination). Note that you can vary the alphas across the brush to make the brush add more of its color in the middle and less on the edges, for an antialiased brush shape. (See "Antialiasing.") Similarly, erasers can be implemented by setting the eraser color to the background color.

- The blending functions that use the source or destination colors GL_DST_COLOR or GL_ONE_MINUS_DST_COLOR for the source factor and GL_SRC_COLOR or GL_ONE_MINUS_SRC_COLOR for the destination factor effectively allow you to modulate each color component individually. This operation is equivalent to applying a simple filter for example, multiplying the red component by 80 percent, the green component by 40 percent, and the blue component by 72 percent would simulate viewing the scene through a photographic filter that blocks 20 percent of red light, 60 percent of green, and 28 percent of blue.
- Suppose you want to draw a picture composed of three translucent surfaces, some obscuring others, and all over a solid background. Assume the farthest surface transmits 80 percent of the color behind it, the next transmits 40 percent, and the closest transmits 90 percent. To compose this picture, draw the background first with the default source and destination factors, and then change the blending factors to GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination). Next, draw the farthest surface with an alpha of 0.2, then the middle surface with an alpha of 0.6, and finally the closest surface with an alpha of 0.1.
- If your system has alpha planes, you can render objects one at a time (including their alpha values), read them back, and then perform interesting matting or compositing operations with the fully rendered objects. (See "Compositing 3D Rendered Images" by Tom Duff, SIGGRAPH 1985 Proceedings, p. 41-44, for examples of this technique.) Note that objects used for picture composition can come from any source they can be rendered using OpenGL commands, rendered using techniques such as ray-tracing or radiosity that are implemented in another graphics library, or obtained by scanning in existing images.
- You can create the effect of a nonrectangular raster image by assigning different alpha values to individual fragments in the image. In most cases, you would assign an alpha of 0 to each "invisible" fragment and an alpha of 1.0 to each opaque fragment. For example, you can draw a polygon in the shape of a tree and apply a texture map of foliage; the viewer can see through parts of the rectangular texture that aren't part of the tree if you've assigned them alpha values of 0. This method, sometimes called *billboarding*, is much faster than creating the tree out of three-dimensional polygons. An example of this technique is shown in Figure 6-1: The tree is a single rectangular polygon that can be rotated about the center of the trunk, as shown by the outlines, so that it's always facing the viewer. (See "Texture Functions" in Chapter 9 for more information about blending textures.)

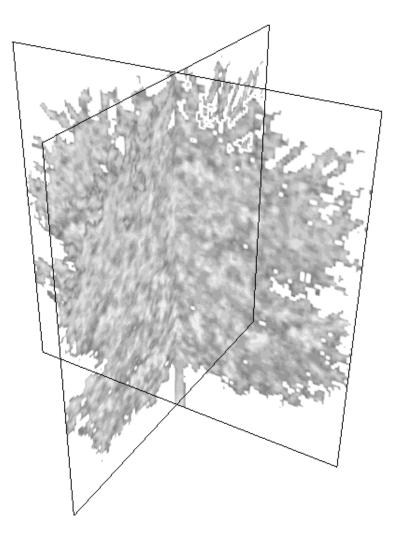


Figure 6-1 : Creating a Nonrectangular Raster Image

• Blending is also used for antialiasing, which is a rendering technique to reduce the jagged appearance of primitives drawn on a raster screen. (See "Antialiasing" for more information.)

A Blending Example

Example 6-1 draws two overlapping colored triangles, each with an alpha of 0.75. Blending is enabled and the source and destination blending factors are set to GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA, respectively.

When the program starts up, a yellow triangle is drawn on the left and then a cyan triangle is drawn on the right so that in the center of the window, where the triangles overlap, cyan is blended with the original yellow. You can change which triangle is drawn first by typing 't' in the window.

Example 6-1 : Blending Example: alpha.c

```
#include <GL/gl.h>
```

```
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
static int leftFirst = GL_TRUE;
/* Initialize alpha blending function. */
static void init(void)
{
  glEnable (GL_BLEND);
  glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glShadeModel (GL_FLAT);
  glClearColor (0.0, 0.0, 0.0, 0.0);
}
static void drawLeftTriangle(void)
/* draw yellow triangle on LHS of screen */
   glBegin (GL_TRIANGLES);
      glColor4f(1.0, 1.0, 0.0, 0.75);
     glVertex3f(0.1, 0.9, 0.0);
     glVertex3f(0.1, 0.1, 0.0);
     glVertex3f(0.7, 0.5, 0.0);
  glEnd();
}
static void drawRightTriangle(void)
/* draw cyan triangle on RHS of screen */
   glBegin (GL_TRIANGLES);
      glColor4f(0.0, 1.0, 1.0, 0.75);
     glVertex3f(0.9, 0.9, 0.0);
     glVertex3f(0.3, 0.5, 0.0);
     glVertex3f(0.9, 0.1, 0.0);
  glEnd();
}
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
   if (leftFirst) {
      drawLeftTriangle();
      drawRightTriangle();
   }
   else {
     drawRightTriangle();
      drawLeftTriangle();
   glFlush();
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
   if (w <= h)
      gluOrtho2D (0.0, 1.0, 0.0, 1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D (0.0, 1.0*(GLfloat)w/(GLfloat)h, 0.0, 1.0);
}
void keyboard(unsigned char key, int x, int y)
```

```
{
   switch (key) {
     case `t':
      case `T':
         leftFirst = !leftFirst;
         glutPostRedisplay();
         break;
      case 27: /* Escape key */
         exit(0);
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT SINGLE | GLUT RGB);
  glutInitWindowSize (200, 200);
  glutCreateWindow (argv[0]);
   init();
  glutReshapeFunc (reshape);
  glutKeyboardFunc (keyboard);
  glutDisplayFunc (display);
  glutMainLoop();
  return 0;
}
```

The order in which the triangles are drawn affects the color of the overlapping region. When the left triangle is drawn first, cyan fragments (the source) are blended with yellow fragments, which are already in the framebuffer (the destination). When the right triangle is drawn first, yellow is blended with cyan. Because the alpha values are all 0.75, the actual blending factors become 0.75 for the source and 1.0 - 0.75 = 0.25 for the destination. In other words, the source fragments are somewhat translucent, but they have more effect on the final color than the destination fragments.

Three-Dimensional Blending with the Depth Buffer

As you saw in the previous example, the order in which polygons are drawn greatly affects the blended result. When drawing three-dimensional translucent objects, you can get different appearances depending on whether you draw the polygons from back to front or from front to back. You also need to consider the effect of the depth buffer when determining the correct order. (See "A Hidden-Surface Removal Survival Kit" in Chapter 5 for an introduction to the depth buffer. Also see "Depth Test" in Chapter 10 for more information.) The depth buffer keeps track of the distance between the viewpoint and the portion of the object occupying a given pixel in a window on the screen; when another candidate color arrives for that pixel, it's drawn only if its object is closer to the viewpoint, in which case its depth value is stored in the depth buffer. With this method, obscured (or hidden) portions of surfaces aren't necessarily drawn and therefore aren't used for blending.

If you want to render both opaque and translucent objects in the same scene, then you want to use the depth buffer to perform hidden-surface removal for any objects that lie behind the opaque objects. If an opaque object hides either a translucent object or another opaque object, you want the depth buffer to eliminate the more distant object. If the translucent object is closer, however, you want to blend it with the opaque object. You can generally figure out the correct order to draw the polygons if everything in the scene is stationary, but the problem can quickly become too hard if either the viewpoint or the object is moving. The solution is to enable depth buffering but make the depth buffer read-only while drawing the translucent objects. First you draw all the opaque objects, with the depth buffer in normal operation. Then you preserve these depth values by making the depth buffer read-only. When the translucent objects are drawn, their depth values are still compared to the values established by the opaque objects, so they aren't drawn if they're behind the opaque ones. If they're closer to the viewpoint, however, they don't eliminate the opaque objects, since the depth-buffer values can't change. Instead, they're blended with the opaque objects. To control whether the depth buffer is writable, use **glDepthMask(**); if you pass GL_FALSE as the argument, the buffer becomes read-only, whereas GL_TRUE restores the normal, writable operation.

Example 6-2 demonstrates how to use this method to draw opaque and translucent three-dimensional objects. In the program, typing 'a' triggers an animation sequence in which a translucent cube moves through an opaque sphere. Pressing the 'r' key resets the objects in the scene to their initial positions. To get the best results when transparent objects overlap, draw the objects from back to front.

Example 6-2: Three-Dimensional Blending: alpha3D.c

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#define MAXZ 8.0
#define MINZ -8.0
#define ZINC 0.4
static float solidZ = MAXZ;
static float transparentZ = MINZ;
static GLuint sphereList, cubeList;
static void init(void)
ł
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 0.15 };
  GLfloat mat_shininess[] = { 100.0 };
  GLfloat position[] = { 0.5, 0.5, 1.0, 0.0 };
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, position);
   glEnable(GL LIGHTING);
   glEnable(GL LIGHT0);
   glEnable(GL_DEPTH_TEST);
   sphereList = glGenLists(1);
   glNewList(sphereList, GL COMPILE);
      glutSolidSphere (0.4, 16, 16);
   glEndList();
   cubeList = glGenLists(1);
   glNewList(cubeList, GL_COMPILE);
      glutSolidCube (0.6);
   glEndList();
}
void display(void)
{
  GLfloat mat_solid[] = { 0.75, 0.75, 0.0, 1.0 };
  GLfloat mat_zero[] = { 0.0, 0.0, 0.0, 1.0 };
```

```
GLfloat mat_transparent[] = { 0.0, 0.8, 0.8, 0.6 };
   GLfloat mat_emission[] = { 0.0, 0.3, 0.3, 0.6 };
   glClear (GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
   glPushMatrix ();
      glTranslatef (-0.15, -0.15, solidZ);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_zero);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_solid);
      glCallList (sphereList);
   glPopMatrix ();
   glPushMatrix ();
      glTranslatef (0.15, 0.15, transparentZ);
      glRotatef (15.0, 1.0, 1.0, 0.0);
      glRotatef (30.0, 0.0, 1.0, 0.0);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_transparent);
      glEnable (GL_BLEND);
      glDepthMask (GL_FALSE);
      glBlendFunc (GL_SRC_ALPHA, GL_ONE);
      glCallList (cubeList);
      glDepthMask (GL TRUE);
      glDisable (GL BLEND);
   glPopMatrix ();
   glutSwapBuffers();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLint) w, (GLint) h);
   glMatrixMode(GL PROJECTION);
   glLoadIdentity();
   if (w \le h)
      glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w,
             1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-1.5*(GLfloat)w/(GLfloat)h,
             1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void animate(void)
ł
   if (solidZ <= MINZ || transparentZ >= MAXZ)
      glutIdleFunc(NULL);
   else {
      solidZ -= ZINC;
      transparentZ += ZINC;
      glutPostRedisplay();
   }
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case `a':
      case `A':
         solidZ = MAXZ;
         transparentZ = MINZ;
         glutIdleFunc(animate);
         break;
      case `r':
```

```
case `R':
         solidZ = MAXZ;
         transparentZ = MINZ;
         glutPostRedisplay();
         break;
      case 27:
        exit(0);
    }
}
int main(int argc, char** argv)
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Antialiasing

You might have noticed in some of your OpenGL pictures that lines, especially nearly horizontal or nearly vertical ones, appear jagged. These jaggies appear because the ideal line is approximated by a series of pixels that must lie on the pixel grid. The jaggedness is called aliasing, and this section describes antialiasing techniques to reduce it. Figure 6-2 shows two intersecting lines, both aliased and antialiased. The pictures have been magnified to show the effect.

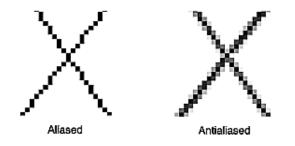


Figure 6-2 : Aliased and Antialiased Lines

Figure 6-3 shows how a diagonal line 1 pixel wide covers more of some pixel squares than others. In fact, when performing antialiasing, OpenGL calculates a *coverage* value for each fragment based on the fraction of the pixel square on the screen that it would cover. The figure shows these coverage values for the line. In RGBA mode, OpenGL multiplies the fragment's alpha value by its coverage. You can then use the resulting alpha value to blend the fragment with the corresponding pixel already in the framebuffer. In color-index mode, OpenGL sets the least significant 4 bits of the color index based on the fragment's coverage (0000 for no coverage and 1111 for complete coverage). It's up to you to load your color map and apply it appropriately to take advantage of this coverage information.

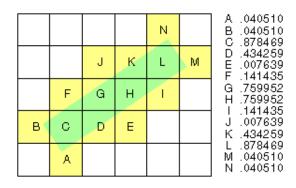


Figure 6-3 : Determining Coverage Values

The details of calculating coverage values are complex, difficult to specify in general, and in fact may vary slightly depending on your particular implementation of OpenGL. You can use the **glHint()** command to exercise some control over the trade-off between image quality and speed, but not all implementations will take the hint.

void glHint(GLenum target, GLenum hint);

Controls certain aspects of OpenGL behavior. The target parameter indicates which behavior is to be controlled; its possible values are shown in Table 6-2. The hint parameter can be GL_FASTEST to indicate that the most efficient option should be chosen, GL_NICEST to indicate the highest-quality option, or GL_DONT_CARE to indicate no preference. The interpretation of hints is implementation-dependent; an implementation can ignore them entirely. (For more information about the relevant topics, see "Antialiasing" for the details on sampling and "Fog" for details on fog.)

The GL_PERSPECTIVE_CORRECTION_HINT target parameter refers to how color values and texture coordinates are interpolated across a primitive: either linearly in screen space (a relatively simple calculation) or in a perspective-correct manner (which requires more computation). Often, systems perform linear color interpolation because the results, while not technically correct, are visually acceptable; however, in most cases textures require perspective-correct interpolation to be visually acceptable. Thus, an implementation can choose to use this parameter to control the method used for interpolation. (See Chapter 3 for a discussion of perspective projection, Chapter 4 for a discussion of color, and Chapter 9 for a discussion of texture mapping.)

Table 6-2 : Values for Use with glHint()

Parameter	Meaning	
GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT	Specify the desired sampling quality of points, lines, or polygons during antialiasing operations	
GL_FOG_HINT	Specifies whether fog calculations are done per pixel (GL_NICEST) or per vertex (GL_FASTEST)	
GL_PERSPECTIVE_CORRECTION_HINT	Specifies the desired quality of color and texture-coordinate interpolation	

Antialiasing Points or Lines

To antialias points or lines, you need to turn on antialiasing with **glEnable**(), passing in GL_POINT_SMOOTH or GL_LINE_SMOOTH, as appropriate. You might also want to provide a quality hint with **glHint**(). (Remember that you can set the size of a point or the width of a line. You can also stipple a line. See "Line Details" in Chapter 2.) Next follow the procedures described in one of the following sections, depending on whether you're in RGBA or color-index mode.

Antialiasing in RGBA Mode

In RGBA mode, you need to enable blending. The blending factors you most likely want to use are GL_SRC_ALPHA (source) and GL_ONE_MINUS_SRC_ALPHA (destination). Alternatively, you can use GL_ONE for the destination factor to make lines a little brighter where they intersect. Now you're ready to draw whatever points or lines you want antialiased. The antialiased effect is most noticeable if you use a fairly high alpha value. Remember that since you're performing blending, you might need to consider the rendering order as described in "Three-Dimensional Blending with the Depth Buffer." However, in most cases, the ordering can be ignored without significant adverse effects. Example 6-3 initializes the necessary modes for antialiasing and then draws two intersecting diagonal lines. When you run this program, press the 'r' key to rotate the lines so that you can see the effect of antialiasing on lines of different slopes. Note that the depth buffer isn't enabled in this example.

Example 6-3 : Antialiased lines: aargb.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <Stdlib.h>
#include <stdlib.h>
#include <stdio.h>
static float rotAngle = 0.;
/* Initialize antialiasing for RGBA mode, including alpha
* blending, hint, and line width. Print out implementation
* specific info on line width granularity and width.
*/
```

```
void init(void)
ł
  GLfloat values[2];
  glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);
   printf ("GL_LINE_WIDTH_GRANULARITY value is %3.1f\n",
      values[0]);
  glGetFloatv (GL_LINE_WIDTH_RANGE, values);
  printf ("GL_LINE_WIDTH_RANGE values are %3.1f %3.1f\n",
      values[0], values[1]);
  glEnable (GL_LINE_SMOOTH);
  glEnable (GL_BLEND);
  glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
  glLineWidth (1.5);
  glClearColor(0.0, 0.0, 0.0, 0.0);
}
/* Draw 2 diagonal lines to form an X */
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f (0.0, 1.0, 0.0);
  glPushMatrix();
  glRotatef(-rotAngle, 0.0, 0.0, 0.1);
  glBegin (GL_LINES);
      glVertex2f (-0.5, 0.5);
      glVertex2f (0.5, -0.5);
   glEnd ();
   glPopMatrix();
  glColor3f (0.0, 0.0, 1.0);
  glPushMatrix();
  glRotatef(rotAngle, 0.0, 0.0, 0.1);
  glBegin (GL LINES);
      glVertex2f (0.5, 0.5);
      glVertex2f (-0.5, -0.5);
   glEnd ();
   glPopMatrix();
  glFlush();
}
void reshape(int w, int h)
  glViewport(0, 0, (GLint) w, (GLint) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
   if (w <= h)
      gluOrtho2D (-1.0, 1.0,
         -1.0*(GLfloat)h/(GLfloat)w, 1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D (-1.0*(GLfloat)w/(GLfloat)h,
         1.0*(GLfloat)w/(GLfloat)h, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void keyboard(unsigned char key, int x, int y)
ł
   switch (key) {
      case `r':
      case `R':
         rotAngle += 20.;
```

```
if (rotAngle >= 360.) rotAngle = 0.;
         glutPostRedisplay();
         break;
      case 27: /* Escape Key */
         exit(0);
         break;
      default:
         break;
    }
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT SINGLE | GLUT RGB);
  glutInitWindowSize (200, 200);
  glutCreateWindow (argv[0]);
   init();
   glutReshapeFunc (reshape);
  glutKeyboardFunc (keyboard);
  glutDisplayFunc (display);
  glutMainLoop();
  return 0;
}
```

Antialiasing in Color-Index Mode

The tricky part about antialiasing in color-index mode is loading and using the color map. Since the last 4 bits of the color index indicate the coverage value, you need to load sixteen contiguous indices with a color ramp from the background color to the object's color. (The ramp has to start with an index value that's a multiple of 16.) Then you clear the color buffer to the first of the sixteen colors in the ramp and draw your points or lines using colors in the ramp. Example 6-4 demonstrates how to construct the color ramp to draw antialiased lines in color-index mode. In this example, two color ramps are created: one contains shades of green and the other shades of blue.

Example 6-4 : Antialiasing in Color-Index Mode: aaindex.c

```
#include <GL/gl.h>
#include <GL/qlu.h>
#include <GL/glut.h>
#include <stdlib.h>
#define RAMPSIZE 16
#define RAMP1START 32
#define RAMP2START 48
static float rotAngle = 0.;
/*
    Initialize antialiasing for color-index mode,
 *
   including loading a green color ramp starting
   at RAMPISTART, and a blue color ramp starting
 *
   at RAMP2START. The ramps must be a multiple of 16.
 * /
void init(void)
{
   int i;
   for (i = 0; i < RAMPSIZE; i++) {</pre>
      GLfloat shade;
      shade = (GLfloat) i/(GLfloat) RAMPSIZE;
      glutSetColor(RAMP1START+(GLint)i, 0., shade, 0.);
      glutSetColor(RAMP2START+(GLint)i, 0., 0., shade);
```

```
}
  glEnable (GL_LINE_SMOOTH);
  glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
  glLineWidth (1.5);
  glClearIndex ((GLfloat) RAMP1START);
}
/* Draw 2 diagonal lines to form an X */
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glIndexi(RAMP1START);
  glPushMatrix();
  glRotatef(-rotAngle, 0.0, 0.0, 0.1);
  glBegin (GL_LINES);
      glVertex2f (-0.5, 0.5);
      glVertex2f (0.5, -0.5);
   glEnd ();
  glPopMatrix();
  glIndexi(RAMP2START);
  glPushMatrix();
  glRotatef(rotAngle, 0.0, 0.0, 0.1);
  glBegin (GL LINES);
      glVertex2f (0.5, 0.5);
      glVertex2f (-0.5, -0.5);
   glEnd ();
   glPopMatrix();
  glFlush();
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
   if (w <= h)
      gluOrtho2D (-1.0, 1.0,
         -1.0*(GLfloat)h/(GLfloat)w, 1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D (-1.0*(GLfloat)w/(GLfloat)h,
         1.0*(GLfloat)w/(GLfloat)h, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
     case `r':
      case `R':
         rotAngle += 20.;
         if (rotAngle >= 360.) rotAngle = 0.;
         glutPostRedisplay();
         break;
      case 27: /* Escape Key */
         exit(0);
         break;
     default:
         break;
    }
}
```

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_INDEX);
    glutInitWindowSize (200, 200);
    glutCreateWindow (argv[0]);
    init();
    glutReshapeFunc (reshape);
    glutKeyboardFunc (keyboard);
    glutDisplayFunc (display);
    glutMainLoop();
    return 0;
}
```

Since the color ramp goes from the background color to the object's color, the antialiased lines look correct only in the areas where they are drawn on top of the background. When the blue line is drawn, it erases part of the green line at the point where the lines intersect. To fix this, you would need to redraw the area where the lines intersect using a ramp that goes from green (the color of the line in the framebuffer) to blue (the color of the line being drawn). However, this requires additional calculations and it is usually not worth the effort since the intersection area is small. Note that this is not a problem in RGBA mode, since the colors of object being drawn are blended with the color already in the framebuffer.

You may also want to enable the depth test when drawing antialiased points and lines in color-index mode. In this example, the depth test is disabled since both of the lines lie in the same *z*-plane. However, if you want to draw a three-dimensional scene, you should enable the depth buffer so that the resulting pixel colors correspond to the "nearest" objects.

The trick described in "Three-Dimensional Blending with the Depth Buffer" can also be used to mix antialiased points and lines with aliased, depth-buffered polygons. To do this, draw the polygons first, then make the depth buffer read-only and draw the points and lines. The points and lines intersect nicely with each other but will be obscured by nearer polygons.

Try This

Take a previous program, such as the robot arm or solar system examples described in "Examples of Composing Several Transformations" in Chapter 3, and draw wireframe objects with antialiasing. Try it in either RGBA or color-index mode. Also try different line widths or point sizes to see their effects.

Antialiasing Polygons

Antialiasing the edges of filled polygons is similar to antialiasing points and lines. When different polygons have overlapping edges, you need to blend the color values appropriately. You can either use the method described in this section, or you can use the accumulation buffer to perform antialiasing for your entire scene. Using the accumulation buffer, which is described in Chapter 10, is easier from your point of view, but it's much more computation-intensive and therefore slower. However, as you'll see, the method described here is rather cumbersome.

Note: If you draw your polygons as points at the vertices or as outlines - that is, by passing GL_POINT or GL_LINE to **glPolygonMode()** - point or line antialiasing is applied, if enabled as described earlier. The rest of this section addresses polygon antialiasing when you're using GL_FILL as the polygon mode.

In theory, you can antialias polygons in either RGBA or color-index mode. However, object intersections affect polygon antialiasing more than they affect point or line antialiasing, so rendering order and blending accuracy become more critical. In fact, they're so critical that if you're antialiasing more than one polygon, you need to order the polygons from front to back and then use **glBlendFunc**() with GL_SRC_ALPHA_SATURATE for the source factor and GL_ONE for the destination factor. Thus, antialiasing polygons in color-index mode normally isn't practical.

To antialias polygons in RGBA mode, you use the alpha value to represent coverage values of polygon edges. You need to enable polygon antialiasing by passing GL_POLYGON_SMOOTH to **glEnable()**. This causes pixels on the edges of the polygon to be assigned fractional alpha values based on their coverage, as though they were lines being antialiased. Also, if you desire, you can supply a value for GL_POLYGON_SMOOTH_HINT.

Now you need to blend overlapping edges appropriately. First, turn off the depth buffer so that you have control over how overlapping pixels are drawn. Then set the blending factors to GL_SRC_ALPHA_SATURATE (source) and GL_ONE (destination). With this specialized blending function, the final color is the sum of the destination color and the scaled source color; the scale factor is the smaller of either the incoming source alpha value or one minus the destination alpha value. This means that for a pixel with a large alpha value, successive incoming pixels have little effect on the final color because one minus the destination alpha is almost zero. With this method, a pixel on the edge of a polygon might be blended eventually with the colors from another polygon that's drawn later. Finally, you need to sort all the polygons in your scene so that they're ordered from front to back before drawing them.

Example 6-5 shows how to antialias filled polygons; clicking the left mouse button toggles the antialiasing on and off. Note that backward-facing polygons are culled and that the alpha values in the color buffer are cleared to zero before any drawing. Pressing the 't' key toggles the antialiasing on and off.

Note: Your color buffer must store alpha values for this technique to work correctly. Make sure you request GLUT_ALPHA and receive a legitimate window.

Example 6-5 : Antialiasing Filled Polygons: aapoly.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
GLboolean polySmooth = GL_TRUE;
static void init(void)
   glCullFace (GL_BACK);
   glEnable (GL_CULL_FACE);
   glBlendFunc (GL_SRC_ALPHA_SATURATE, GL_ONE);
   glClearColor (0.0, 0.0, 0.0, 0.0);
}
#define NFACE 6
#define NVERT 8
void drawCube(GLdouble x0, GLdouble x1, GLdouble y0,
              GLdouble y1, GLdouble z0, GLdouble z1)
{
   static GLfloat v[8][3];
```

```
static GLfloat c[8][4] = {
    {0.0, 0.0, 0.0, 1.0}, {1.0, 0.0, 0.0, 1.0},
    {0.0, 1.0, 0.0, 1.0}, {1.0, 1.0, 0.0, 1.0},
    {0.0, 0.0, 1.0, 1.0}, {1.0, 0.0, 1.0, 1.0},
    {0.0, 1.0, 1.0, 1.0}, {1.0, 1.0, 1.0, 1.0},
}
   };
/* indices of front, top, left, bottom, right, back faces */
   static GLubyte indices[NFACE][4] = {
       \{4, 5, 6, 7\}, \{2, 3, 7, 6\}, \{0, 4, 7, 3\},\
       \{0, 1, 5, 4\}, \{1, 5, 6, 2\}, \{0, 3, 2, 1\}
   };
   v[0][0] = v[3][0] = v[4][0] = v[7][0] = x0;
   v[1][0] = v[2][0] = v[5][0] = v[6][0] = x1;
   v[0][1] = v[1][1] = v[4][1] = v[5][1] = y0;
   v[2][1] = v[3][1] = v[6][1] = v[7][1] = y1;
   v[0][2] = v[1][2] = v[2][2] = v[3][2] = z0;
   v[4][2] = v[5][2] = v[6][2] = v[7][2] = z1;
#ifdef GL_VERSION_1_1
   glEnableClientState (GL_VERTEX_ARRAY);
   glEnableClientState (GL_COLOR_ARRAY);
   glVertexPointer (3, GL FLOAT, 0, v);
   glColorPointer (4, GL_FLOAT, 0, c);
   glDrawElements(GL QUADS, NFACE*4, GL UNSIGNED BYTE, indices);
   glDisableClientState (GL_VERTEX_ARRAY);
   glDisableClientState (GL_COLOR_ARRAY);
#else
   printf ("If this is GL Version 1.0, ");
   printf ("vertex arrays are not supported.\n");
  exit(1);
#endif
}
/*
   Note: polygons must be drawn from front to back
 * for proper blending.
 */
void display(void)
{
   if (polySmooth) {
      glClear (GL_COLOR_BUFFER_BIT);
      glEnable (GL_BLEND);
      glEnable (GL_POLYGON_SMOOTH);
      glDisable (GL_DEPTH_TEST);
   }
   else {
      qlClear (GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
      glDisable (GL_BLEND);
      glDisable (GL_POLYGON_SMOOTH);
      glEnable (GL_DEPTH_TEST);
   }
   glPushMatrix ();
      glTranslatef (0.0, 0.0, -8.0);
      glRotatef (30.0, 1.0, 0.0, 0.0);
      glRotatef (60.0, 0.0, 1.0, 0.0);
      drawCube(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5);
   glPopMatrix ();
   glFlush ();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
```

```
glMatrixMode(GL PROJECTION);
   glLoadIdentity();
   gluPerspective(30.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
   glMatrixMode(GL MODELVIEW);
   glLoadIdentity();
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case `t':
      case `T':
         polySmooth = !polySmooth;
         glutPostRedisplay();
         break;
      case 27:
         exit(0); /* Escape key */
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT SINGLE | GLUT RGB
                         GLUT ALPHA | GLUT DEPTH);
   glutInitWindowSize(200, 200);
   glutCreateWindow(argv[0]);
   init ();
   glutReshapeFunc (reshape);
  glutKeyboardFunc (keyboard);
  glutDisplayFunc (display);
  glutMainLoop();
  return 0;
}
```

Fog

Computer images sometimes seem unrealistically sharp and well defined. Antialiasing makes an object appear more realistic by smoothing its edges. Additionally, you can make an entire image appear more natural by adding fog, which makes objects fade into the distance. *Fog* is a general term that describes similar forms of atmospheric effects; it can be used to simulate haze, mist, smoke, or pollution. (See Plate 9.) Fog is essential in visual-simulation applications, where limited visibility needs to be approximated. It's often incorporated into flight-simulator displays.

When fog is enabled, objects that are farther from the viewpoint begin to fade into the fog color. You can control the density of the fog, which determines the rate at which objects fade as the distance increases, as well as the fog's color. Fog is available in both RGBA and color-index modes, although the calculations are slightly different in the two modes. Since fog is applied after matrix transformations, lighting, and texturing are performed, it affects transformed, lit, and textured objects. Note that with large simulation programs, fog can improve performance, since you can choose not to draw objects that would be too fogged to be visible.

All types of geometric primitives can be fogged, including points and lines. Using the fog effect on points and lines is also called depth-cuing (as shown in Plate 2) and is popular in molecular modeling and other applications.

Using Fog

Using fog is easy. You enable it by passing GL_FOG to **glEnable**(), and you choose the color and the equation that controls the density with **glFog***(). If you want, you can supply a value for GL_FOG_HINT with **glHint**(), as described on Table 6-2. Example 6-6 draws five red spheres, each at a different distance from the viewpoint. Pressing the 'f' key selects among the three different fog equations, which are described in the next section.

Example 6-6 : Five Fogged Spheres in RGBA Mode: fog.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
static GLint fogMode;
static void init(void)
{
  GLfloat position[] = { 0.5, 0.5, 3.0, 0.0 };
  glEnable(GL DEPTH TEST);
  glLightfv(GL_LIGHT0, GL_POSITION, position);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
     GLfloat mat[3] = {0.1745, 0.01175, 0.01175};
     glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
     mat[0] = 0.61424; mat[1] = 0.04136; mat[2] = 0.04136;
     glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
     mat[0] = 0.727811; mat[1] = 0.626959; mat[2] = 0.626959;
     glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
     glMaterialf (GL_FRONT, GL_SHININESS, 0.6*128.0);
   }
   glEnable(GL_FOG);
      GLfloat fogColor[4] = \{0.5, 0.5, 0.5, 1.0\};
     foqMode = GL EXP;
     glFoqi (GL FOG MODE, foqMode);
     glFogfv (GL FOG COLOR, fogColor);
     glFogf (GL FOG DENSITY, 0.35);
     glHint (GL_FOG_HINT, GL_DONT_CARE);
     glFogf (GL_FOG_START, 1.0);
     glFogf (GL_FOG_END, 5.0);
   glClearColor(0.5, 0.5, 0.5, 1.0); /* fog color */
}
static void renderSphere (GLfloat x, GLfloat y, GLfloat z)
ł
  glPushMatrix();
  glTranslatef (x, y, z);
  glutSolidSphere(0.4, 16, 16);
   glPopMatrix();
}
```

```
/* display() draws 5 spheres at different z positions.
 */
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  renderSphere (-2., -0.5, -1.0);
  renderSphere (-1., -0.5, -2.0);
  renderSphere (0., -0.5, -3.0);
  renderSphere (1., -0.5, -4.0);
  renderSphere (2., -0.5, -5.0);
  glFlush();
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
   if (w \le h)
      glOrtho (-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
         2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-2.5*(GLfloat)w/(GLfloat)h,
         2.5*(GLfloat)w/(GLfloat)h, -2.5, 2.5, -10.0, 10.0);
   glMatrixMode(GL MODELVIEW);
  glLoadIdentity ();
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
     case `f':
      case `F':
         if (foqMode == GL EXP) {
            foqMode = GL EXP2;
            printf ("Fog mode is GL EXP2\n");
         }
         else if (fogMode == GL_EXP2) {
            fogMode = GL_LINEAR;
            printf ("Fog mode is GL_LINEAR\n");
         }
         else if (fogMode == GL_LINEAR) {
            fogMode = GL_EXP;
            printf ("Fog mode is GL_EXP\n");
         }
         glFogi (GL_FOG_MODE, fogMode);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
     default:
         break;
   }
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutCreateWindow(argv[0]);
   init();
  glutReshapeFunc (reshape);
   glutKeyboardFunc (keyboard);
```

```
glutDisplayFunc (display);
glutMainLoop();
return 0;
```

Fog Equations

}

Fog blends a fog color with an incoming fragment's color using a fog blending factor. This factor, f, is computed with one of these three equations and then clamped to the range [0,1].

 $f = e^{(density.z)} (GL_EXP)$ $f = e^{(density.z)^2} (GL_EXP2)$ $f = \frac{end - z}{end - start} (GL_LINEAR)$

In these three equations, z is the eye-coordinate distance between the viewpoint and the fragment center. The values for *density*, *start*, and *end* are all specified with **glFog*(**). The f factor is used differently, depending on whether you're in RGBA mode or color-index mode, as explained in the next subsections.

void glFog{if}(GLenum pname, TYPE param);

void glFog{if}v(GLenum pname, TYPE *params);
Sets the parameters and function for calculating fog. If pname is GL_FOG_MODE, then param is either GL_EXP (the default), GL_EXP2, or GL_LINEAR to select one of the three fog factors. If pname is GL_FOG_DENSITY, GL_FOG_START, or GL_FOG_END, then param is (or points to, with the vector version of the command) a value for density, start, or end in the equations. (The default values are 1, 0, and 1, respectively.) In RGBA mode, pname can be GL_FOG_COLOR, in which case params points to four values that specify the fog's RGBA color values. The corresponding value for pname in color-index mode is GL_FOG_INDEX, for which param is a single value specifying the fog's color index.

Figure 6-4 plots the fog-density equations for various values of the parameters.

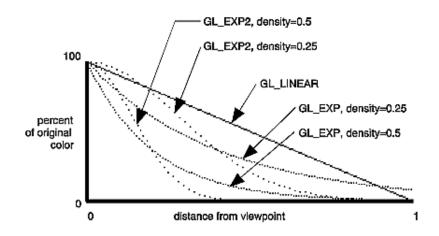


Figure 6-4 : Fog-Density Equations

Fog in RGBA Mode

In RGBA mode, the fog factor f is used as follows to calculate the final fogged color:

C = f Ci + (1 - f) Cf

where Ci represents the incoming fragment's RGBA values and Cf the fog-color values assigned with GL_FOG_COLOR.

Fog in Color-Index Mode

In color-index mode, the final fogged color index is computed as follows:

I = Ii + (1 - f) If

where Ii is the incoming fragment's color index and If is the fog's color index as specified with GL_FOG_INDEX.

To use fog in color-index mode, you have to load appropriate values in a color ramp. The first color in the ramp is the color of the object without fog, and the last color in the ramp is the color of the completely fogged object. You probably want to use **glClearIndex(**) to initialize the background color index so that it corresponds to the last color in the ramp; this way, totally fogged objects blend into the background. Similarly, before objects are drawn, you should call **glIndex*(**) and pass in the index of the first color in the ramp (the unfogged color). Finally, to apply fog to different colored objects in the scene, you need to create several color ramps and call **glIndex*(**) before each object is drawn to set the current color index to the start of each color ramp. Example 6-7 illustrates how to initialize appropriate conditions and then apply fog in color-index mode.

Example 6-7 : Fog in Color-Index Mode: fogindex.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
  Initialize color map and fog. Set screen clear color
/*
  to end of color ramp.
 * /
#define NUMCOLORS 32
#define RAMPSTART 16
static void init(void)
   int i;
  glEnable(GL_DEPTH_TEST);
   for (i = 0; i < NUMCOLORS; i++) {
     GLfloat shade;
      shade = (GLfloat) (NUMCOLORS-i)/(GLfloat) NUMCOLORS;
      glutSetColor (RAMPSTART + i, shade, shade, shade);
   }
   glEnable(GL FOG);
   glFogi (GL FOG MODE, GL LINEAR);
   glFogi (GL FOG INDEX, NUMCOLORS);
   glFogf (GL_FOG_START, 1.0);
```

```
glFogf (GL_FOG_END, 6.0);
   glHint (GL_FOG_HINT, GL_NICEST);
   glClearIndex((GLfloat) (NUMCOLORS+RAMPSTART-1));
}
static void renderSphere (GLfloat x, GLfloat y, GLfloat z)
ł
  glPushMatrix();
  glTranslatef (x, y, z);
  glutWireSphere(0.4, 16, 16);
  glPopMatrix();
}
/* display() draws 5 spheres at different z positions.
 * /
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glIndexi (RAMPSTART);
  renderSphere (-2., -0.5, -1.0);
  renderSphere (-1., -0.5, -2.0);
  renderSphere (0., -0.5, -3.0);
  renderSphere (1., -0.5, -4.0);
  renderSphere (2., -0.5, -5.0);
  glFlush();
}
void reshape(int w, int h)
{
  glViewport(0, 0, w, h);
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
   if (w \le h)
      glOrtho (-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
         2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho (-2.5*(GLfloat)w/(GLfloat)h,
         2.5*(GLfloat)w/(GLfloat)h, -2.5, 2.5, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity ();
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
   }
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_INDEX | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutCreateWindow(argv[0]);
   init();
  glutReshapeFunc (reshape);
  glutKeyboardFunc (keyboard);
  glutDisplayFunc (display);
  glutMainLoop();
  return 0;
```

Polygon Offset

If you want to highlight the edges of a solid object, you might try to draw the object with polygon mode GL_FILL and then draw it again, but in a different color with polygon mode GL_LINE. However, because lines and filled polygons are not rasterized in exactly the same way, the depth values generated for pixels on a line are usually not the same as the depth values for a polygon edge, even between the same two vertices. The highlighting lines may fade in and out of the coincident polygons, which is sometimes called "stitching" and is visually unpleasant.

The visual unpleasantness can be eliminated by using polygon offset, which adds an appropriate offset to force coincident *z* values apart to cleanly separate a polygon edge from its highlighting line. (The stencil buffer, described in "Stencil Test" in Chapter 10, can also be used to eliminate stitching. However, polygon offset is almost always faster than stenciling.) Polygon offset is also useful for applying decals to surfaces, rendering images with hidden-line removal. In addition to lines and filled polygons, this technique can also be used with points.

There are three different ways to turn on polygon offset, one for each type of polygon rasterization mode: GL_FILL, GL_LINE, or GL_POINT. You enable the polygon offset by passing the appropriate parameter to **glEnable**(), either GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE, or GL_POLYGON_OFFSET_POINT. You must also call **glPolygonMode**() to set the current polygon rasterization method.

void glPolygonOffset(GLfloat factor, GLfloat units);

When enabled, the depth value of each fragment is added to a calculated offset value. The offset is added before the depth test is performed and before the depth value is written into the depth buffer. The offset value o is calculated by: o = m * factor + r * unitswhere m is the maximum depth slope of the polygon and r is the smallest value guaranteed to produce a resolvable difference in window coordinate depth values. The value r is an implementation-specific constant.

To achieve a nice rendering of the highlighted solid object without visual artifacts, you can either add a positive offset to the solid object (push it away from you) or a negative offset to the wireframe (pull it towards you). The big question is: "How much offset is enough?" Unfortunately, the offset required depends upon various factors, including the depth slope of each polygon and the width of the lines in the wireframe.

OpenGL calculates the depth slope (see Figure 6-5) of a polygon for you, but it's important that you understand what the depth slope is, so that you choose a reasonable value for *factor*. The depth slope is the change in z (depth) values divided by the change in either x or y coordinates, as you traverse a polygon. The depth values are in window coordinates, clamped to the range [0, 1]. To estimate the maximum depth slope of a polygon (m in the offset equation), use this formula:

$$m = max\left\{ \left| \frac{\partial V}{\partial s} \right|, \left| \frac{\partial V}{\partial t} \right| \right\}$$

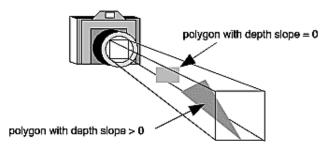


Figure 6-5 : Polygons and Their Depth Slopes

For polygons that are parallel to the near and far clipping planes, the depth slope is zero. For the polygons in your scene with a depth slope near zero, only a small, constant offset is needed. To create a small, constant offset, you can pass *factor*=0.0 and *units*=1.0 to **glPolygonOffset**().

For polygons that are at a great angle to the clipping planes, the depth slope can be significantly greater than zero, and a larger offset may be needed. Small, non-zero values for *factor*, such as 0.75 or 1.0, are probably enough to generate distinct depth values and eliminate the unpleasant visual artifacts.

Example 6-8 shows a portion of code, where a display list (which presumably draws a solid object) is first rendered with lighting, the default GL_FILL polygon mode, and polygon offset with *factor* of 1.0 and *units* of 1.0. These values ensure that the offset is enough for all polygons in your scene, regardless of depth slope. (These values may actually be a little more offset than the minimum needed, but too much offset is less noticeable than too little.) Then, to highlight the edges of the first object, the object is rendered as an unlit wireframe with the offset disabled.

Example 6-8 : Polygon Offset to Eliminate Visual Artifacts: polyoff.c

glEnable(GL_LIGHTING); glEnable(GL_LIGHT0); glEnable(GL_POLYGON_OFFSET_FILL); glPolygonOffset(1.0, 1.0); glCallList (list); glDisable(GL_POLYGON_OFFSET_FILL); glDisable(GL_LIGHTING); glDisable(GL_LIGHT0); glColor3f (1.0, 1.0, 1.0); glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

In a few situations, the simplest values for *factor* and *units* (1.0 and 1.0) aren't the answers. For instance, if the width of the lines that are highlighting the edges are greater than one, then increasing the value of *factor* may be necessary. Also, since depth values are unevenly transformed into window coordinates when using perspective projection (see "The Transformed Depth Coordinate" in Chapter 3), less offset is needed for polygons that are closer to the near clipping plane, and more offset is needed for polygons that are further away. Once again, experimenting with the value of *factor* may be warranted.

Chapter 7 Display Lists

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand how display lists can be used along with commands in immediate mode to organize your data and improve performance
- Maximize performance by knowing how and when to use display lists

A *display list* is a group of OpenGL commands that have been stored for later execution. When a display list is invoked, the commands in it are executed in the order in which they were issued. Most OpenGL commands can be either stored in a display list or issued in *immediate mode*, which causes them to be executed immediately. You can freely mix immediate-mode programming and display lists within a single program. The programming examples you've seen so far have used immediate mode. This chapter discusses what display lists are and how best to use them. It has the following major sections:

- "Why Use Display Lists?" explains when to use display lists.
- "An Example of Using a Display List" gives a brief example, showing the basic commands for using display lists.
- "Display-List Design Philosophy" explains why certain design choices were made (such as making display lists uneditable) and what performance optimizations you might expect to see when using display lists.
- "Creating and Executing a Display List" discusses in detail the commands for creating, executing, and deleting display lists.
- "Executing Multiple Display Lists" shows how to execute several display lists in succession, using a small character set as an example.
- "Managing State Variables with Display Lists" illustrates how to use display lists to save and restore OpenGL commands that set state variables.

Why Use Display Lists?

Display lists may improve performance since you can use them to store OpenGL commands for

later execution. It is often a good idea to cache commands in a display list if you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times. Using display lists, you can define the geometry and/or state changes once and execute them multiple times.

To see how you can use display lists to store geometry just once, consider drawing a tricycle. The two wheels on the back are the same size but are offset from each other. The front wheel is larger than the back wheels and also in a different location. An efficient way to render the wheels on the tricycle would be to store the geometry for one wheel in a display list then execute the list three times. You would need to set the modelview matrix appropriately each time before executing the list to calculate the correct size and location for the wheels.

When running OpenGL programs remotely to another machine on the network, it is especially important to cache commands in a display list. In this case, the server is a different machine than the host. (See "What Is OpenGL?" in Chapter 1 for a discussion of the OpenGL client-server model.) Since display lists are part of the server state and therefore reside on the server machine, you can reduce the cost of repeatedly transmitting that data over a network if you store repeatedly used commands in a display list.

When running locally, you can often improve performance by storing frequently used commands in a display list. Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form that is more compatible with the graphics hardware or software. (See "Display-List Design Philosophy" for a detailed discussion of these optimizations.)

An Example of Using a Display List

A display list is a convenient and efficient way to name and organize a set of OpenGL commands. For example, suppose you want to draw a torus and view it from different angles. The most efficient way to do this would be to store the torus in a display list. Then whenever you want to change the view, you would change the modelview matrix and execute the display list to draw the torus. Example 7-1 illustrates this.

Example 7-1 : Creating a Display List: torus.c

```
#include <GL/ql.h>
#include <GL/qlu.h>
#include <stdio.h>
#include <math.h>
#include <GL/qlut.h>
#include <stdlib.h>
GLuint theTorus;
/* Draw a torus */
static void torus(int numc, int numt)
   int i, j, k;
   double s, t, x, y, z, twopi;
   twopi = 2 * (double)M_PI;
   for (i = 0; i < numc; i++) {</pre>
      glBegin(GL_QUAD_STRIP);
      for (j = 0; j <= numt; j++)</pre>
         for (k = 1; k \ge 0; k--)
```

```
s = (i + k) % numc + 0.5;
            t = j % numt;
            x = (1+.1*cos(s*twopi/numc))*cos(t*twopi/numt);
            y = (1+.1*cos(s*twopi/numc))*sin(t*twopi/numt);
            z = .1 * sin(s * twopi / numc);
            glVertex3f(x, y, z);
         }
      }
      glEnd();
   }
}
/* Create display list with Torus and initialize state*/
static void init(void)
{
  theTorus = glGenLists (1);
  glNewList(theTorus, GL_COMPILE);
  torus(8, 25);
  glEndList();
  glShadeModel(GL_FLAT);
  glClearColor(0.0, 0.0, 0.0, 0.0);
}
void display(void)
{
  glClear(GL COLOR BUFFER BIT);
  glColor3f (1.0, 1.0, 1.0);
  glCallList(theTorus);
  glFlush();
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
  gluPerspective(30, (GLfloat) w/(GLfloat) h, 1.0, 100.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0, 0, 10, 0, 0, 0, 0, 1, 0);
}
/* Rotate about x-axis when "x" typed; rotate about y-axis
  when "y" typed; "i" returns torus to original view */
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
  case `x':
   case `X':
      glRotatef(30.,1.0,0.0,0.0);
      glutPostRedisplay();
     break;
   case 'y':
   case `Y':
     glRotatef(30.,0.0,1.0,0.0);
      glutPostRedisplay();
     break;
   case `i':
   case `I':
      glLoadIdentity();
      gluLookAt(0, 0, 10, 0, 0, 0, 0, 1, 0);
      glutPostRedisplay();
     break;
```

```
case 27:
      exit(0);
      break;
   }
}
int main(int argc, char **argv)
   glutInitWindowSize(200, 200);
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Let's start by looking at **init**(). It creates a display list for the torus and initializes the viewing matrices and other rendering state. Note that the routine for drawing a torus (**torus**()) is bracketed by **glNewList**() and **glEndList**(), which defines a display list. The argument *listName* for **glNewList**() is an integer index, generated by **glGenLists**(), that uniquely identifies this display list.

The user can rotate the torus about the x- or y-axis by pressing the 'x' or 'y' key when the window has focus. Whenever this happens, the callback function **keyboard**() is called, which concatenates a 30-degree rotation matrix (about the x- or y-axis) with the current modelview matrix. Then **glutPostRedisplay**() is called, which will cause **glutMainLoop**() to call **display**() and render the torus after other events have been processed. When the 'i' key is pressed, **keyboard**() restores the initial modelview matrix and returns the torus to its original location.

The **display()** function is very simple: It clears the window and then calls **glCallList()** to execute the commands in the display list. If we hadn't used display lists, **display()** would have to reissue the commands to draw the torus each time it was called.

A display list contains only OpenGL commands. In Example 7-1, only the **glBegin**(), **glVertex**(), and **glEnd**() calls are stored in the display list. The parameters for the calls are *evaluated*, and their values are copied into the display list when it is created. All the trigonometry to create the torus is done only once, which should increase rendering performance. However, the values in the display list can't be changed later. And once a command has been stored in a list it is not possible to remove it. Neither can you add any new commands to the list after it has been defined. You can delete the entire display list and create a new one, but you can't edit it.

Note: Display lists also work well with GLU commands, since those operations are ultimately broken down into low-level OpenGL commands, which can easily be stored in display lists. Use of display lists with GLU is particularly important for optimizing performance of GLU tessellators and NURBS.

Display-List Design Philosophy

To optimize performance, an OpenGL display list is a cache of commands rather than a dynamic database. In other words, once a display list is created, it can't be modified. If a display list were modifiable, performance could be reduced by the overhead required to search through the display

list and perform memory management. As portions of a modifiable display list were changed, memory allocation and deallocation might lead to memory fragmentation. Any modifications that the OpenGL implementation made to the display-list commands in order to make them more efficient to render would need to be redone. Also, the display list may be difficult to access, cached somewhere over a network or a system bus.

The way in which the commands in a display list are optimized may vary from implementation to implementation. For example, a command as simple as **glRotate***() might show a significant improvement if it's in a display list, since the calculations to produce the rotation matrix aren't trivial (they can involve square roots and trigonometric functions). In the display list, however, only the final rotation matrix needs to be stored, so a display-list rotation command can be executed as fast as the hardware can execute **glMultMatrix***(). A sophisticated OpenGL implementation might even concatenate adjacent transformation commands into a single matrix multiplication.

Although you're not guaranteed that your OpenGL implementation optimizes display lists for any particular uses, the execution of display lists isn't slower than executing the commands contained within them individually. There is some overhead, however, involved in jumping to a display list. If a particular list is small, this overhead could exceed any execution advantage. The most likely possibilities for optimization are listed next, with references to the chapters where the topics are discussed.

- Matrix operations (Chapter 3). Most matrix operations require OpenGL to compute inverses. Both the computed matrix and its inverse might be stored by a particular OpenGL implementation in a display list.
- Raster bitmaps and images (Chapter 8). The format in which you specify raster data isn't likely to be one that's ideal for the hardware. When a display list is compiled, OpenGL might transform the data into the representation preferred by the hardware. This can have a significant effect on the speed of raster character drawing, since character strings usually consist of a series of small bitmaps.
- Lights, material properties, and lighting models (Chapter 5). When you draw a scene with complex lighting conditions, you might change the materials for each item in the scene. Setting the materials can be slow, since it might involve significant calculations. If you put the material definitions in display lists, these calculations don't have to be done each time you switch materials, since only the results of the calculations need to be stored; as a result, rendering lit scenes might be faster. (See "Encapsulating Mode Changes" for more details on using display lists to change such values as lighting conditions.)
- Textures (Chapter 9). You might be able to maximize efficiency when defining textures by compiling them into a display list, since the display list may allow the texture image to be cached in dedicated texture memory. Then the texture image would not have to be recopied each time it was needed. Also, the hardware texture format might differ from the OpenGL format, and the conversion can be done at display-list compile time rather than during display.

In OpenGL version 1.0, the display list is the primary method to manage textures. However, if the OpenGL implementation that you are using is version 1.1 or greater, then you should store the texture in a texture object instead. (Some version 1.0 implementations have a vendor-specific extension to support texture objects. If your implementation supports texture objects, you are encouraged to use them.)

• Polygon stipple patterns (Chapter 2).

Some of the commands to specify the properties listed here are context-sensitive, so you need to take this into account to ensure optimum performance. For example, when

GL_COLOR_MATERIAL is enabled, some of the material properties will track the current color. (See Chapter 5.) Any **glMaterial***() calls that set the same material properties are ignored.

It may improve performance to store state settings with geometry. For example, suppose you want to apply a transformation to some geometric objects and then draw the result. Your code may look like this:

```
glNewList(1, GL_COMPILE);
draw_some_geometric_objects();
glEndList();
glLoadMatrix(M);
glCallList(1);
```

However, if the geometric objects are to be transformed in the same way each time, it is better to store the matrix in the display list. For example, if you were to write your code as follows, some implementations may be able to improve performance by transforming the objects when they are defined instead of each time they are drawn:

```
glNewList(1, GL_COMPILE);
glLoadMatrix(M);
draw_some_geometric_objects();
glEndList();
glCallList(1);
```

A more likely situation occurs when rendering images. As you will see in Chapter 8, you can modify pixel transfer state variables and control the way images and bitmaps are rasterized. If the commands that set these state variables precede the definition of the image or bitmap in the display list, the implementation may be able to perform some of the operations ahead of time and cache the result.

Remember that display lists have some disadvantages. Very small lists may not perform well since there is some overhead when executing a list. Another disadvantage is the immutability of the contents of a display list. To optimize performance, an OpenGL display list can't be changed and its contents can't be read. If the application needs to maintain data separately from the display list (for example, for continued data processing), then a lot of additional memory may be required.

Creating and Executing a Display List

As you've already seen, **glNewList(**) and **glEndList(**) are used to begin and end the definition of a display list, which is then invoked by supplying its identifying index with **glCallList(**). In Example 7-2, a display list is created in the **init(**) routine. This display list contains OpenGL commands to draw a red triangle. Then in the **display(**) routine, the display list is executed ten times. In addition, a line is drawn in immediate mode. Note that the display list allocates memory to store the commands and the values of any necessary variables.

Example 7-2 : Using a Display List: list.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
GLuint listName;
static void init (void)
ł
   listName = glGenLists (1);
   glNewList (listName, GL_COMPILE);
      glColor3f (1.0, 0.0, 0.0); /* current color red */
      glBegin (GL_TRIANGLES);
      glVertex2f (0.0, 0.0);
      glVertex2f (1.0, 0.0);
      glVertex2f (0.0, 1.0);
      glEnd ();
      glTranslatef (1.5, 0.0, 0.0); /* move position */
   glEndList ();
   glShadeModel (GL_FLAT);
}
static void drawLine (void)
ł
  glBegin (GL_LINES);
  glVertex2f (0.0, 0.5);
  glVertex2f (15.0, 0.5);
  glEnd ();
}
void display(void)
{
  GLuint i;
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (0.0, 1.0, 0.0); /*
                                   current color green
                                                         * /
                               /*
   for (i = 0; i < 10; i++)
                                                         */
                                   draw 10 triangles
      glCallList (listName);
   drawLine (); /* is this line green? NO!
                                                */
                 /* where is the line drawn? */
   glFlush ();
}
void reshape(int w, int h)
ł
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
   if (w \le h)
      gluOrtho2D (0.0, 2.0, -0.5 * (GLfloat) h/(GLfloat) w,
         1.5 * (GLfloat) h/(GLfloat) w);
   else
      gluOrtho2D (0.0, 2.0*(GLfloat) w/(GLfloat) h, -0.5, 1.5);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void keyboard(unsigned char key, int x, int y)
ł
   switch (key) {
      case 27:
         exit(0);
   }
}
```

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(650, 50);
    glutCreateWindow(argv[0]);
    init ();
    glutReshapeFunc (reshape);
    glutReshapeFunc (keyboard);
    glutDisplayFunc (display);
    glutDisplayFunc (display);
    return 0;
}
```

The **glTranslatef**() routine in the display list alters the position of the next object to be drawn. Without it, calling the display list twice would just draw the triangle on top of itself. The **drawLine**() routine, which is called in immediate mode, is also affected by the ten **glTranslatef**() calls that precede it. So if you call transformation commands within a display list, don't forget to take into account the effect those commands will have later in your program.

Only one display list can be created at a time. In other words, you must eventually follow **glNewList()** with **glEndList()** to end the creation of a display list before starting another one. As you might expect, calling **glEndList()** without having started a display list generates the error GL_INVALID_OPERATION. (See "Error Handling" in Chapter 14 for more information about processing errors.)

Naming and Creating a Display List

Each display list is identified by an integer index. When creating a display list, you want to be careful that you don't accidentally choose an index that's already in use, thereby overwriting an existing display list. To avoid accidental deletions, use **glGenLists()** to generate one or more unused indices.

GLuint glGenLists(GLsizei range);

Allocates range number of contiguous, previously unallocated display-list indices. The integer returned is the index that marks the beginning of a contiguous block of empty display-list indices. The returned indices are all marked as empty and used, so subsequent calls to **glGenLists()** don't return these indices until they're deleted. Zero is returned if the requested number of indices isn't available, or if range is zero.

In the following example, a single index is requested, and if it proves to be available, it's used to create a new display list:

```
listIndex = glGenLists(1);
if (listIndex != 0) {
   glNewList(listIndex,GL_COMPILE);
        ...
   glEndList();
}
```

Note: Zero is not a valid display-list index.

void glNewList (GLuint list, GLenum mode);

Specifies the start of a display list. OpenGL routines that are called subsequently (until **glEndList()** is called to end the display list) are stored in a display list, except for a few

restricted OpenGL routines that can't be stored. (Those restricted routines are executed immediately, during the creation of the display list.) list is a nonzero positive integer that uniquely identifies the display list. The possible values for mode are GL_COMPILE and GL_COMPILE_AND_EXECUTE. Use GL_COMPILE if you don't want the OpenGL commands executed as they're placed in the display list; to cause the commands to be executed immediately as well as placed in the display list for later use, specify GL_COMPILE_AND_EXECUTE.

void glEndList (void);

Marks the end of a display list.

When a display list is created it is stored with the current OpenGL context. Thus, when the context is destroyed, the display list is also destroyed. Some windowing systems allow multiple contexts to share display lists. In this case, the display list is destroyed when the last context in the share group is destroyed.

What's Stored in a Display List

When you're building a display list, only the values for expressions are stored in the list. If values in an array are subsequently changed, the display-list values don't change. In the following code fragment, the display list contains a command to set the current RGBA color to black (0.0, 0.0, 0.0). The subsequent change of the value of the *color_vector* array to red (1.0, 0.0, 0.0) has no effect on the display list because the display list contains the values that were in effect when it was created.

```
GLfloat color_vector[3] = {0.0, 0.0, 0.0};
glNewList(1, GL_COMPILE);
glColor3fv(color_vector);
glEndList();
color_vector[0] = 1.0;
```

Not all OpenGL commands can be stored and executed from within a display list. For example, commands that set client state and commands that retrieve state values aren't stored in a display list. (Many of these commands are easily identifiable because they return values in parameters passed by reference or return a value directly.) If these commands are called when making a display list, they're executed immediately.

Here are the OpenGL commands that aren't stored in a display list (also, note that **glNewList**() generates an error if it's called while you're creating a display list). Some of these commands haven't been described yet; you can look in the index to see where they're discussed.

glColorPointer()	glFlush()	glNormalPointer()
glDeleteLists()	glGenLists()	glPixelStore()
glDisableClientState()	glGet*()	glReadPixels()
glEdgeFlagPointer()	glIndexPointer()	glRenderMode()
glEnableClientState()	glInterleavedArrays()	glSelectBuffer()
glFeedbackBuffer()	glIsEnabled()	glTexCoordPointer()

glFinish()

To understand more clearly why these commands can't be stored in a display list, remember that when you're using OpenGL across a network, the client may be on one machine and the server on another. After a display list is created, it resides with the server, so the server can't rely on the client for any information related to the display list. If querying commands, such as **glGet***() or **glIs***(), were allowed in a display list, the calling program would be surprised at random times by data returned over the network. Without parsing the display list as it was sent, the calling program wouldn't know where to put the data. Thus, any command that returns a value can't be stored in a display list. In addition, commands that change client state, such as **glPixelStore**(), **glSelectBuffer**(), and the commands to define vertex arrays, can't be stored in a display list.

The operation of some OpenGL commands depends upon client state. For example, the vertex array specification routines (such as glVertexPointer()glColorPointer(), and glInterleavedArrays()) set client state pointers and cannot be stored in a display list. glArrayElement(), glDrawArrays(), and glDrawElements() send data to the server state to construct primitives from elements in the enabled arrays, so these operations can be stored in a display list. (See "Vertex Arrays" in Chapter 2.) The vertex array data stored in this display list is obtained by dereferencing data from the pointers, not by storing the pointers themselves. Therefore, subsequent changes to the data in the vertex arrays will not affect the definition of the primitive in the display list.

In addition, any commands that use the pixel storage modes use the modes that are in effect when they are placed in the display list. (See "Controlling Pixel-Storage Modes" in Chapter 8.) Other routines that rely upon client state - such as **glFlush()** and **glFinish()** - can't be stored in a display list because they depend upon the client state that is in effect when they are executed.

Executing a Display List

After you've created a display list, you can execute it by calling **glCallList(**). Naturally, you can execute the same display list many times, and you can mix calls to execute display lists with calls to perform immediate-mode graphics, as you've already seen.

void glCallList (GLuint list);

This routine executes the display list specified by list. The commands in the display list are executed in the order they were saved, just as if they were issued without using a display list. If list hasn't been defined, nothing happens.

You can call **glCallList**() from anywhere within a program, as long as an OpenGL context that can access the display list is active (that is, the context that was active when the display list was created or a context in the same share group). A display list can be created in one routine and executed in a different one, since its index uniquely identifies it. Also, there is no facility to save the contents of a display list into a data file, nor a facility to create a display list from a file. In this sense, a display list is designed for temporary use.

Hierarchical Display Lists

You can create a *hierarchical display list*, which is a display list that executes another display list by calling **glCallList()** between a **glNewList()** and **glEndList()** pair. A hierarchical display list is useful for an object made of components, especially if some of those components are used more than once. For example, this is a display list that renders a bicycle by calling other display lists to

render parts of the bicycle:

```
glNewList(listIndex,GL_COMPILE);
glCallList(handlebars);
glCallList(frame);
glTranslatef(1.0,0.0,0.0);
glCallList(wheel);
glTranslatef(3.0,0.0,0.0);
glCallList(wheel);
glEndList();
```

To avoid infinite recursion, there's a limit on the nesting level of display lists; the limit is at least 64, but it might be higher, depending on the implementation. To determine the nesting limit for your implementation of OpenGL, call

glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data);

OpenGL allows you to create a display list that calls another list that hasn't been created yet. Nothing happens when the first list calls the second, undefined one.

You can use a hierarchical display list to approximate an editable display list by wrapping a list around several lower-level lists. For example, to put a polygon in a display list while allowing yourself to be able to easily edit its vertices, you could use the code in Example 7-3.

Example 7-3 : Hierarchical Display List

```
glNewList(1,GL_COMPILE);
   glVertex3f(v1);
glEndList();
glNewList(2,GL COMPILE);
   glVertex3f(v2);
qlEndList();
glNewList(3,GL COMPILE);
   glVertex3f(v3);
glEndList();
glNewList(4,GL COMPILE);
   glBegin(GL POLYGON);
      glCallList(1);
      glCallList(2);
      glCallList(3);
   glEnd();
glEndList();
```

To render the polygon, call display list number 4. To edit a vertex, you need only recreate the single display list corresponding to that vertex. Since an index number uniquely identifies a display list, creating one with the same index as an existing one automatically deletes the old one. Keep in mind that this technique doesn't necessarily provide optimal memory usage or peak performance, but it's acceptable and useful in some cases.

Managing Display List Indices

So far, we've recommended the use of **glGenLists(**) to obtain unused display-list indices. If you insist upon avoiding **glGenLists(**), then be sure to use **glIsList(**) to determine whether a specific index is in use.

GLboolean gllsList(GLuint list);

You can explicitly delete a specific display list or a contiguous range of lists with **glDeleteLists**(). Using **glDeleteLists**() makes those indices available again.

void glDeleteLists(GLuint list, GLsizei range);

Deletes range display lists, starting at the index specified by list. An attempt to delete a list that has never been created is ignored.

Executing Multiple Display Lists

OpenGL provides an efficient mechanism to execute several display lists in succession. This mechanism requires that you put the display-list indices in an array and call **glCallLists**(). An obvious use for such a mechanism occurs when display-list indices correspond to meaningful values. For example, if you're creating a font, each display-list index might correspond to the ASCII value of a character in that font. To have several such fonts, you would need to establish a different initial display-list index for each font. You can specify this initial index by using **glListBase**() before calling **glCallLists**().

void glListBase(GLuint base);

Specifies the offset that's added to the display-list indices in **glCallLists()** to obtain the final display-list indices. The default display-list base is 0. The list base has no effect on **glCallList()**, which executes only one display list or on **glNewList()**.

void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);

Executes n display lists. The indices of the lists to be executed are computed by adding the offset indicated by the current display-list base (specified with **glListBase(**)) to the signed integer values in the array pointed to by lists.

The type parameter indicates the data type of the values in lists. It can be set to GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT,

GL_UNSIGNED_INT, or GL_FLOAT, indicating that lists should be treated as an array of bytes, unsigned bytes, shorts, unsigned shorts, integers, unsigned integers, or floats, respectively. Type can also be GL_2_BYTES, GL_3_BYTES, or GL_4_BYTES, in which case sequences of 2, 3, or 4 bytes are read from lists and then shifted and added together, byte by byte, to calculate the display-list offset. The following algorithm is used (where byte[0] is the start of a byte sequence).

```
/* b = 2, 3, or 4; bytes are numbered 0, 1, 2, 3 in array */
offset = 0;
for (i = 0; i < b; i++) {
    offset = offset << 8;
    offset += byte[i];
}
index = offset + listbase;</pre>
```

For multiple-byte data, the highest-order data comes first as bytes are taken from the array in order.

As an example of the use of multiple display lists, look at the program fragments in Example 7-4 taken from the full program in Example 7-5. This program draws characters with a stroked font (a set of letters made from line segments). The routine **initStrokedFont**() sets up the display-list

indices for each letter so that they correspond with their ASCII values.

Example 7-4 : Defining Multiple Display Lists

```
void initStrokedFont(void)
   GLuint base;
   base = glGenLists(128);
   glListBase(base);
   glNewList(base+'A', GL_COMPILE);
      drawLetter(Adata); glEndList();
   glNewList(base+'E', GL_COMPILE);
      drawLetter(Edata); glEndList();
   glNewList(base+'P', GL_COMPILE);
      drawLetter(Pdata); glEndList();
   glNewList(base+'R', GL_COMPILE);
      drawLetter(Rdata); glEndList();
   glNewList(base+'S', GL_COMPILE);
      drawLetter(Sdata); glEndList();
   glNewList(base+' ', GL_COMPILE);
                                        /* space character */
      glTranslatef(8.0, 0.0, 0.0);
   glEndList();
}
```

The **glGenLists**() command allocates 128 contiguous display-list indices. The first of the contiguous indices becomes the display-list base. A display list is made for each letter; each display-list index is the sum of the base and the ASCII value of that letter. In this example, only a few letters and the space character are created.

After the display lists have been created, **glCallLists**() can be called to execute the display lists. For example, you can pass a character string to the subroutine **printStrokedString**():

```
void printStrokedString(GLbyte *s)
{
    GLint len = strlen(s);
    glCallLists(len, GL_BYTE, s);
}
```

The ASCII value for each letter in the string is used as the offset into the display-list indices. The current list base is added to the ASCII value of each letter to determine the final display-list index to be executed. The output produced by Example 7-5 is shown in Figure 7-1.



Figure 7-1 : Stroked Font That Defines the Characters A, E, P, R, S

Example 7-5 : Multiple Display Lists to Define a Stroked Font: stroke.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <string.h>
```

```
#define PT 1
#define STROKE 2
#define END 3
typedef struct charpoint {
   GLfloat x, y;
   int
          type;
} CP;
CP Adata[] = {
   { 0, 0, PT}, {0, 9, PT}, {1, 10, PT}, {4, 10, PT},
   {5, 9, PT}, {5, 0, STROKE}, {0, 5, PT}, {5, 5, END}
};
CP Edata[] = {
   {5, 0, PT}, {0, 0, PT}, {0, 10, PT}, {5, 10, STROKE},
   \{0, 5, PT\}, \{4, 5, END\}
};
CP Pdata[] = {
   {0, 0, PT}, {0, 10, PT}, {4, 10, PT}, {5, 9, PT}, {5, 6, PT},
   \{4, 5, PT\}, \{0, 5, END\}
};
CP Rdata[] = {
   {0, 0, PT}, {0, 10, PT}, {4, 10, PT}, {5, 9, PT}, {5, 6, PT},
   \{4, 5, PT\}, \{0, 5, STROKE\}, \{3, 5, PT\}, \{5, 0, END\}
};
CP Sdata[] = {
   {0, 1, PT}, {1, 0, PT}, {4, 0, PT}, {5, 1, PT}, {5, 4, PT}, {4, 5, PT}, {1, 5, PT}, {0, 6, PT}, {0, 9, PT}, {1, 10, PT},
   {4, 10, PT}, {5, 9, END}
};
/* drawLetter() interprets the instructions from the array
 * for that letter and renders the letter with line segments.
 */
static void drawLetter(CP *1)
ł
   glBegin(GL_LINE_STRIP);
   while (1) {
      switch (l->type) {
          case PT:
             glVertex2fv(&l->x);
             break;
          case STROKE:
             glVertex2fv(&l->x);
             glEnd();
             glBegin(GL_LINE_STRIP);
             break;
          case END:
             glVertex2fv(&l->x);
             glEnd();
             glTranslatef(8.0, 0.0, 0.0);
             return;
      }
      1++;
   }
}
/* Create a display list for each of 6 characters
                                                             */
static void init (void)
{
```

```
GLuint base;
   glShadeModel (GL FLAT);
   base = glGenLists (128);
   glListBase(base);
   glNewList(base+'A', GL_COMPILE); drawLetter(Adata);
   glEndList();
   glNewList(base+'E', GL_COMPILE); drawLetter(Edata);
   glEndList();
   glNewList(base+'P', GL_COMPILE); drawLetter(Pdata);
  glEndList();
  glNewList(base+'R', GL_COMPILE); drawLetter(Rdata);
  glEndList();
  glNewList(base+'S', GL_COMPILE); drawLetter(Sdata);
  glEndList();
  glNewList(base+' `, GL_COMPILE);
  glTranslatef(8.0, 0.0, 0.0); glEndList();
}
char *test1 = "A SPARE SERAPE APPEARS AS";
char *test2 = "APES PREPARE RARE PEPPERS";
static void printStrokedString(char *s)
  GLsizei len = strlen(s);
  glCallLists(len, GL_BYTE, (GLbyte *)s);
}
void display(void)
{
  glClear(GL COLOR BUFFER BIT);
  glColor3f(1.0, 1.0, 1.0);
  glPushMatrix();
  glScalef(2.0, 2.0, 2.0);
  glTranslatef(10.0, 30.0, 0.0);
  printStrokedString(test1);
  glPopMatrix();
  glPushMatrix();
  glScalef(2.0, 2.0, 2.0);
  glTranslatef(10.0, 13.0, 0.0);
  printStrokedString(test2);
  glPopMatrix();
   glFlush();
}
void reshape(int w, int h)
ł
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
     case ` `:
        glutPostRedisplay();
         break;
      case 27:
         exit(0);
   }
}
```

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (440, 120);
    glutCreateWindow (argv[0]);
    init ();
    glutReshapeFunc(reshape);
    glutReshapeFunc(keyboard);
    glutDisplayFunc(display);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Managing State Variables with Display Lists

A display list can contain calls that change the value of OpenGL state variables. These values change as the display list is executed, just as if the commands were called in immediate mode and the changes persist after execution of the display list is completed. As previously seen in Example 7-2 and in Example 7-6, which follows, the changes to the current color and current matrix made during the execution of the display list remain in effect after it has been called.

Example 7-6 : Persistence of State Changes after Execution of a Display List

```
glNewList(listIndex,GL_COMPILE);
glColor3f(1.0, 0.0, 0.0);
glBegin(GL_POLYGON);
glVertex2f(0.0,0.0);
glVertex2f(1.0,0.0);
glVertex2f(0.0,1.0);
glEnd();
glTranslatef(1.5,0.0,0.0);
```

So if you now call the following sequence, the line drawn after the display list is drawn with red as the current color and translated by an additional (1.5, 0.0, 0.0):

```
glCallList(listIndex);
glBegin(GL_LINES);
glVertex2f(2.0,-1.0);
glVertex2f(1.0,0.0);
glEnd();
```

Sometimes you want state changes to persist, but other times you want to save the values of state variables before executing a display list and then restore these values after the list has executed. Remember that you cannot use **glGet*()** in a display list, so you must use another way to query and store the values of state variables.

You can use **glPushAttrib**() to save a group of state variables and **glPopAttrib**() to restore the values when you're ready for them. To save and restore the current matrix, use **glPushMatrix**() and **glPopMatrix**() as described in "Manipulating the Matrix Stacks" in Chapter 3. These push and pop routines can be legally cached in a display list. To restore the state variables in Example 7-6, you might use the code shown in Example 7-7.

Example 7-7 : Restoring State Variables within a Display List

```
glNewList(listIndex,GL_COMPILE);
  glPushMatrix();
  glPushAttrib(GL_CURRENT_BIT);
  glColor3f(1.0, 0.0, 0.0);
  glBegin(GL_POLYGON);
    glVertex2f(0.0,0.0);
    glVertex2f(1.0,0.0);
    glVertex2f(0.0,1.0);
  glEnd();
  glTranslatef(1.5,0.0,0.0);
  glPopAttrib();
  glPopMatrix();
  glEndList();
```

If you use the display list from Example 7-7, which restores values, the code in Example 7-8 draws a green, untranslated line. With the display list in Example 7-6, which doesn't save and restore values, the line is drawn red, and its position is translated ten times (1.5, 0.0, 0.0).

Example 7-8 : The Display List May or May Not Affect drawLine()

```
void display(void)
{
    GLint i;
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0); /* set current color to green */
    for (i = 0; i < 10; i++)
        glCallList(listIndex); /* display list called 10 times */
        drawLine(); /* how and where does this line appear? */
        glFlush();
}</pre>
```

Encapsulating Mode Changes

You can use display lists to organize and store groups of commands to change various modes or set various parameters. When you want to switch from one group of settings to another, using display lists might be more efficient than making the calls directly, since the settings might be cached in a format that matches the requirements of your graphics system.

Display lists may be more efficient than immediate mode for switching among various lighting, lighting-model, and material-parameter settings. You might also use display lists for stipple patterns, fog parameters, and clipping-plane equations. In general, you'll find that executing display lists is at least as fast as making the relevant calls directly, but remember that some overhead is involved in jumping to a display list.

Example 7-9 shows how to use display lists to switch among three different line stipples. First, you call **glGenLists**() to allocate a display list for each stipple pattern and create a display list for each pattern. Then, you use **glCallList**() to switch from one stipple pattern to another.

Example 7-9 : Display Lists for Mode Changes

```
GLuint offset;
offset = glGenLists(3);
glNewList (offset, GL_COMPILE);
    glDisable (GL_LINE_STIPPLE);
glEndList ();
```

```
glNewList (offset+1, GL_COMPILE);
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x0F0F);
glEndList ();
glNewList (offset+2, GL COMPILE);
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, 0x1111);
glEndList ();
#define drawOneLine(x1,y1,x2,y2) glBegin(GL_LINES); \
    glVertex2f ((x1),(y1)); glVertex2f ((x2),(y2)); glEnd();
glCallList (offset);
drawOneLine (50.0, 125.0, 350.0, 125.0);
glCallList (offset+1);
drawOneLine (50.0, 100.0, 350.0, 100.0);
glCallList (offset+2);
drawOneLine (50.0, 75.0, 350.0, 75.0);
```

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 8 Drawing Pixels, Bitmaps, Fonts, and Images

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Position and draw bitmapped data
- Read pixel data (bitmaps and images) from the framebuffer into processor memory and from memory into the framebuffer
- Copy pixel data from one color buffer to another, or to another location in the same buffer
- Magnify or reduce an image as it's written to the framebuffer
- Control pixel-data formatting and perform other transformations as the data is moved to and from the framebuffer

So far, most of the discussion in this guide has concerned the rendering of geometric data - points, lines, and polygons. Two other important classes of data that can be rendered by OpenGL are

- Bitmaps, typically used for characters in fonts
- Image data, which might have been scanned in or calculated

Both bitmaps and image data take the form of rectangular arrays of pixels. One difference between them is that a bitmap consists of a single bit of information about each pixel, and image data typically includes several pieces of data per pixel (the complete red, green, blue, and alpha color components, for example). Also, bitmaps are like masks in that they're used to overlay another image, but image data simply overwrites or is blended with whatever data is in the framebuffer.

This chapter describes how to draw pixel data (bitmaps and images) from processor memory to the framebuffer and how to read pixel data from the framebuffer into processor memory. It also describes how to copy pixel data from one position to another, either from one buffer to another or within a single buffer. This chapter contains the following major sections:

- "Bitmaps and Fonts" describes the commands for positioning and drawing bitmapped data. Such data may describe a font.
- "Images" presents the basic information about drawing, reading and copying pixel data.
- "Imaging Pipeline" describes the operations that are performed on images and bitmaps when they are read from the framebuffer and when they are written to the framebuffer.

- "Reading and Drawing Pixel Rectangles" covers all the details of how pixel data is stored in memory and how to transform it as it's moved into or out of memory.
- "Tips for Improving Pixel Drawing Rates" lists tips for getting better performance when drawing pixel rectangles.

In most cases, the necessary pixel operations are simple, so the first three sections might be all you need to read for your application. However, pixel manipulation can be complex - there are many ways to store pixel data in memory, and you can apply any of several transformations to pixels as they're moved to and from the framebuffer. These details are the subject of the fourth section of this chapter. Most likely, you'll want to read this section only when you actually need to make use of the information. The last section provides useful tips to get the best performance when rendering bitmaps and images.

Bitmaps and Fonts

A bitmap is a rectangular array of 0s and 1s that serves as a drawing mask for a corresponding rectangular portion of the window. Suppose you're drawing a bitmap and that the current raster color is red. Wherever there's a 1 in the bitmap, the corresponding pixel is replaced by a red pixel (or combined with a red pixel, depending on which per-fragment operations are in effect. (See "Testing and Operating on Fragments" in Chapter 10.) If there's a 0 in the bitmap, the contents of the pixel are unaffected. The most common use of bitmaps is for drawing characters on the screen.

OpenGL provides only the lowest level of support for drawing strings of characters and manipulating fonts. The commands **glRasterPos***() and **glBitmap**() position and draw a single bitmap on the screen. In addition, through the display-list mechanism, you can use a sequence of character codes to index into a corresponding series of bitmaps representing those characters. (See Chapter 7 for more information about display lists.) You'll have to write your own routines to provide any other support you need for manipulating bitmaps, fonts, and strings of characters.

Consider Example 8-1, which draws the character F three times on the screen. Figure 8-1 shows the F as a bitmap and its corresponding bitmap data.

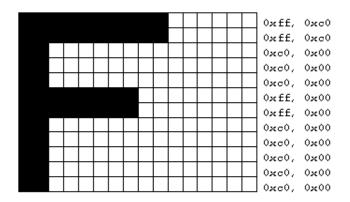


Figure 8-1 : Bitmapped F and Its Data

Example 8-1 : Drawing a Bitmapped Character: drawf.c

```
#include <GL/ql.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
GLubyte rasters[24] = {
   0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
   0xff, 0x00, 0xff, 0x00, 0xc0, 0x00, 0xc0, 0x00, 0xc0, 0x00,
   0xff, 0xc0, 0xff, 0xc0};
void init(void)
  glPixelStorei (GL UNPACK ALIGNMENT, 1);
  glClearColor (0.0, 0.0, 0.0, 0.0);
}
void display(void)
  glClear(GL COLOR BUFFER BIT);
  glColor3f (1.0, 1.0, 1.0);
  glRasterPos2i (20, 20);
  glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
  glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
  glBitmap (10, 12, 0.0, 0.0, 11.0, 0.0, rasters);
  glFlush();
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho (0, w, 0, h, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
     case 27:
         exit(0);
   }
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize(100, 100);
  glutInitWindowPosition(100, 100);
  glutCreateWindow(argv[0]);
   init();
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

In Figure 8-1, note that the visible part of the F character is at most 10 bits wide. Bitmap data is always stored in chunks that are multiples of 8 bits, but the width of the actual bitmap doesn't have to be a multiple of 8. The bits making up a bitmap are drawn starting from the lower-left corner: First, the bottom row is drawn, then the next row above it, and so on. As you can tell from the code,

the bitmap is stored in memory in this order - the array of rasters begins with 0xc0, 0x00, 0xc0, 0x00 for the bottom two rows of the F and continues to 0xff, 0xc0, 0xff, 0xc0 for the top two rows.

The commands of interest in this example are **glRasterPos2i**() and **glBitmap**(); they're discussed in detail in the next section. For now, ignore the call to **glPixelStorei**(); it describes how the bitmap data is stored in computer memory. (See "Controlling Pixel-Storage Modes" for more information.)

The Current Raster Position

The current raster position is the origin where the next bitmap (or image) is to be drawn. In the F example, the raster position was set by calling **glRasterPos***() with coordinates (20, 20), which is where the lower-left corner of the F was drawn:

glRasterPos2i(20, 20);

void glRasterPos{234}{sifd}(TYPE x, TYPE y, TYPE z, TYPE w); void glRasterPos{234}{sifd}v(TYPE *coords);

Sets the current raster position. The x, y, z, and w arguments specify the coordinates of the raster position. If the vector form of the function is used, the coords array contains the coordinates of the raster position. If **glRasterPos2*()** is used, z is implicitly set to zero and w is implicitly set to one; similarly, with **glRasterPos3*()**, w is set to one.

The coordinates of the raster position are transformed to screen coordinates in exactly the same way as coordinates supplied with a **glVertex***() command (that is, with the modelview and perspective matrices). After transformation, they either define a valid spot in the viewport, or they're clipped out because the coordinates were outside the viewing volume. If the transformed point is clipped out, the current raster position is invalid.

Note: If you want to specify the raster position in screen coordinates, you'll want to make sure you've specified the modelview and projection matrices for simple 2D rendering, with something like this sequence of commands, where *width* and *height* are also the size (in pixels) of the viewport:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, (GLfloat) width, 0.0, (GLfloat) height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

To obtain the current raster position, you can use the query command **glGetFloatv**() with $GL_CURRENT_RASTER_POSITION$ as the first argument. The second argument should be a pointer to an array that can hold the (*x*, *y*, *z*, *w*) values as floating-point numbers. Call **glGetBooleanv**() with $GL_CURRENT_RASTER_POSITION_VALID$ as the first argument to determine whether the current raster position is valid.

Drawing the Bitmap

Once you've set the desired raster position, you can use the glBitmap() command to draw the data.

void **glBitmap**(GLsizei width, GLsizei height, GLfloat xbo, GLfloat ybo, GLfloat xbi, GLfloat ybi, const GLubyte *bitmap); Draws the bitmap specified by bitmap, which is a pointer to the bitmap image. The origin of the bitmap is placed at the current raster position. If the current raster position is invalid, nothing is drawn, and the raster position remains invalid. The width and height arguments indicate the width and height, in pixels, of the bitmap. The width need not be a multiple of 8, although the data is stored in unsigned characters of 8 bits each. (In the F example, it wouldn't matter if there were garbage bits in the data beyond the tenth bit; since **glBitmap**() was called with a width of 10, only 10 bits of the row are rendered.) Use xbo and ybo to define the origin of the bitmap (positive values move the origin up and to the right of the raster position; negative values move it down and to the left); xbi and ybi indicate the x and y increments that are added to the raster position after the bitmap is rasterized (see Figure 8-2).

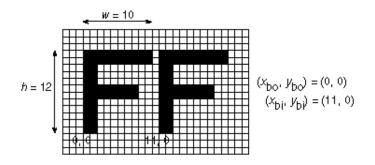


Figure 8-2 : Bitmap and Its Associated Parameters

Allowing the origin of the bitmap to be placed arbitrarily makes it easy for characters to extend below the origin (typically used for characters with descenders, such as g, j, and y), or to extend beyond the left of the origin (used for various swash characters, which have extended flourishes, or for characters in fonts that lean to the left).

After the bitmap is drawn, the current raster position is advanced by *x*bi and *y*bi in the *x*- and *y*-directions, respectively. (If you just want to advance the current raster position without drawing anything, call **glBitmap**() with the *bitmap* parameter set to NULL and with the *width* and *height* set to zero.) For standard Latin fonts, *y*bi is typically 0.0 and *x*bi is positive (since successive characters are drawn from left to right). For Hebrew, where characters go from right to left, the *x*bi values would typically be negative. Fonts that draw successive characters vertically in columns would use zero for *x*bi and nonzero values for *y*bi. In Figure 8-2, each time the F is drawn, the current raster position advances by 11 pixels, allowing a 1-pixel space between successive characters.

Since *x*bo, *y*bo, *x*bi, and *y*bi are floating-point values, characters need not be an integral number of pixels apart. Actual characters are drawn on exact pixel boundaries, but the current raster position is kept in floating point so that each character is drawn as close as possible to where it belongs. For example, if the code in the F example was modified so that *x*bi is 11.5 instead of 12, and if more characters were drawn, the space between letters would alternate between 1 and 2 pixels, giving the best approximation to the requested 1.5-pixel space.

Note: You can't rotate bitmap fonts because the bitmap is always drawn aligned to the *x* and *y* framebuffer axes.

Choosing a Color for the Bitmap

You are familiar with using **glColor***() and **glIndex***() to set the current color or index to draw geometric primitives. The same commands are used to set different state variables, GL_CURRENT_RASTER_COLOR and GL_CURRENT_RASTER_INDEX, for rendering bitmaps. The raster color state variables are set when **glRasterPos***() is called, which can lead to a trap. In the following sequence of code, what is the color of the bitmap?

```
glColor3f(1.0, 1.0, 1.0); /* white */
glRasterPos3fv(position);
glColor3f(1.0, 0.0, 0.0); /* red */
glBitmap(....);
```

The bitmap is white! The GL_CURRENT_RASTER_COLOR is set to white when **glRasterPos3fv**() is called. The second call to **glColor3f**() changes the value of GL_CURRENT_COLOR for future geometric rendering, but the color used to render the bitmap is unchanged.

To obtain the current raster color or index, you can use the query commands **glGetFloatv()** or **glGetIntegerv()** with GL_CURRENT_RASTER_COLOR or GL_CURRENT_RASTER_INDEX as the first argument.

Fonts and Display Lists

Display lists are discussed in general terms in Chapter 7. However, a few of the display-list management commands have special relevance for drawing strings of characters. As you read this section, keep in mind that the ideas presented here apply equally well to characters that are drawn using bitmap data and those drawn using geometric primitives (points, lines, and polygons). (See "Executing Multiple Display Lists" in Chapter 7 for an example of a geometric font.)

A font typically consists of a set of characters, where each character has an identifying number (usually the ASCII code) and a drawing method. For a standard ASCII character set, the capital letter A is number 65, B is 66, and so on. The string "DAB" would be represented by the three indices 68, 65, 66. In the simplest approach, display-list number 65 draws an A, number 66 draws a B, and so on. Then to draw the string 68, 65, 66, just execute the corresponding display lists.

You can use the command glCallLists() in just this way:

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);
```

The first argument, *n*, indicates the number of characters to be drawn, *type* is usually GL_BYTE, and *lists* is an array of character codes.

Since many applications need to draw character strings in multiple fonts and sizes, this simplest approach isn't convenient. Instead, you'd like to use 65 as A no matter what font is currently active. You could force font 1 to encode A, B, and C as 1065, 1066, 1067, and font 2 as 2065, 2066, 2067, but then any numbers larger than 256 would no longer fit in an 8-bit byte. A better solution is to add an offset to every entry in the string and to choose the display list. In this case, font 1 has A, B, and C represented by 1065, 1066, and 1067, and in font 2, they might be 2065, 2066, and 2067. Then to draw characters in font 1, set the offset to 1000 and draw display lists 65, 66, and 67. To draw that same string in font 2, set the offset to 2000 and draw the same lists.

To set the offset, use the command **glListBase(**). For the preceding examples, it should be called with 1000 or 2000 as the (only) argument. Now what you need is a contiguous list of unused

display-list numbers, which you can obtain from glGenLists():

GLuint glGenLists(GLsizei range);

This function returns a block of *range* display-list identifiers. The returned lists are all marked as "used" even though they're empty, so that subsequent calls to **glGenLists**() never return the same lists (unless you've explicitly deleted them previously). Therefore, if you use 4 as the argument and if **glGenLists**() returns 81, you can use display-list identifiers 81, 82, 83, and 84 for your characters. If **glGenLists**() can't find a block of unused identifiers of the requested length, it returns 0. (Note that the command **glDeleteLists**() makes it easy to delete all the lists associated with a font in a single operation.)

Most American and European fonts have a small number of characters (fewer than 256), so it's easy to represent each character with a different code that can be stored in a single byte. Asian fonts, among others, may require much larger character sets, so a byte-per-character encoding is impossible. OpenGL allows strings to be composed of 1-, 2-, 3-, or 4-byte characters through the *type* parameter in **glCallLists**(). This parameter can have any of the following values:

GL_BYTE GL_UNSIGNED_BYTE

GL_SHORT GL_UNSIGNED_SHORT

GL_INT GL_UNSIGNED_INT

GL_FLOAT GL_2_BYTES

GL_3_BYTES GL_4_BYTES

(See "Executing Multiple Display Lists" in Chapter 7 for more information about these values.)

Defining and Using a Complete Font

The **glBitmap()** command and the display-list mechanism described in the previous section make it easy to define a raster font. In Example 8-2, the upper-case characters of an ASCII font are defined. In this example, each character has the same width, but this is not always the case. Once the characters are defined, the program prints the message "THE QUICK BROWN FOX JUMPS OVER A LAZY DOG".

The code in Example 8-2 is similar to the F example, except that each character's bitmap is stored in its own display list. The display list identifier, when combined with the offset returned by **glGenLists**(), is equal to the ASCII code for the character.

Example 8-2 : Drawing a Complete Font: font.c

```
{0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xff, 0xc3, 0xc3, 0xc3, 0x66, 0x3c, 0x1
        0x00, 0x00, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xf
        0x00, 0x00, 0x7e, 0xe7, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc7, 0x7
        0x00, 0x00, 0xfc, 0xce, 0xc7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0xce, 0xf
        0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xc0, 0xc0, 0xf
        0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfc, 0xc0, 0xc0, 0xf
        0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xcf, 0xc0, 0xc0, 0xc0, 0xc0, 0xe7, 0x7
        0x00, 0x00, 0xc3, 0xc3
        0x00, 0x00, 0x7e, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x7e
        0x00, 0x00, 0x7c, 0xee, 0xc6, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06, 0x06
        0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xe0, 0xf0, 0xd8, 0xcc, 0xc6, 0xc
        0x00, 0x00, 0xff, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xc
        0x00, 0x00, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc4, 0xff, 0xff, 0xe7, 0xc
        0x00, 0x00, 0xc7, 0xc7, 0xcf, 0xcf, 0xdf, 0xdb, 0xfb, 0xf3, 0xf3, 0xe3, 0xe
        0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc7, 0x7
       (0x00, 0x00, 0xc0, 0xc0, 0xc0, 0xc0, 0xc0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xf
       \{0x00, 0x00, 0x3f, 0x6e, 0xdf, 0xdb, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0x6f, 0x3, 0xc3, 0xc3, 0x6f, 0x3, 0xc3, 0xc3, 0xc3, 0x6f, 0x3, 0xc3, 
        0x00, 0x00, 0xc3, 0xc6, 0xcc, 0xd8, 0xf0, 0xfe, 0xc7, 0xc3, 0xc3, 0xc7, 0xf
        0x00, 0x00, 0x7e, 0xe7, 0x03, 0x03, 0x07, 0x7e, 0xe0, 0xc0, 0xc0, 0xe7, 0x7
        0x00, 0x00, 0x18, 0x18
        0x00, 0x00, 0x7e, 0xe7, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3
        0x00, 0x00, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc3, 0xc3, 0xc3, 0xc3, 0xc3, 0xc
        0x00, 0x00, 0xc3, 0xe7, 0xff, 0xff, 0xdb, 0xdb, 0xc3, 0xc3, 0xc3, 0xc
        0x00, 0x00, 0xc3, 0x66, 0x66, 0x3c, 0x3c, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc
       {0x00, 0x00, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x3c, 0x3c, 0x66, 0x66, 0xc
       {0x00, 0x00, 0xff, 0xc0, 0xc0, 0x60, 0x30, 0x7e, 0x0c, 0x06, 0x03, 0x6;
};
GLuint fontOffset;
void makeRasterFont(void)
    GLuint i, j;
    glPixelStorei(GL UNPACK ALIGNMENT, 1);
     fontOffset = glGenLists (128);
     for (i = 0, j = A'; i < 26; i++, j++)
          glNewList(fontOffset + j, GL COMPILE);
          glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, letters[i]);
          glEndList();
    glNewList(fontOffset + ` `, GL_COMPILE);
     glBitmap(8, 13, 0.0, 2.0, 10.0, 0.0, space);
     glEndList();
void init(void)
     glShadeModel (GL_FLAT);
    makeRasterFont();
void printString(char *s)
    glPushAttrib (GL_LIST_BIT);
    glListBase(fontOffset);
    glCallLists(strlen(s), GL_UNSIGNED_BYTE, (GLubyte *) s);
    glPopAttrib ();
/* Everything above this line could be in a library
  * that defines a font. To make it work, you've got
 * to call makeRasterFont() before you start making
 * calls to printString().
 * /
```

{

ł

{

```
void display(void)
ł
   GLfloat white[3] = { 1.0, 1.0, 1.0 };
   glClear(GL_COLOR_BUFFER_BIT);
  glColor3fv(white);
  glRasterPos2i(20, 60);
  printString("THE QUICK BROWN FOX JUMPS");
  glRasterPos2i(20, 40);
  printString("OVER A LAZY DOG");
  glFlush ();
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho (0.0, w, 0.0, h, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);
}
void keyboard(unsigned char key, int x, int y)
   switch (key) {
     case 27:
         exit(0);
   }
}
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT SINGLE | GLUT RGB);
  glutInitWindowSize(300, 100);
  glutInitWindowPosition (100, 100);
  glutCreateWindow(argv[0]);
  init();
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Images

An image is similar to a bitmap, but instead of containing only a single bit for each pixel in a rectangular region of the screen, an image can contain much more information. For example, an image can contain a complete (R, G, B, A) color stored at each pixel. Images can come from several sources, such as

- A photograph that's digitized with a scanner
- An image that was first generated on the screen by a graphics program using the graphics hardware and then read back, pixel by pixel
- A software program that generated the image in memory pixel by pixel

The images you normally think of as pictures come from the color buffers. However, you can read or write rectangular regions of pixel data from or to the depth buffer or the stencil buffer. (See Chapter 10 for an explanation of these other buffers.)

In addition to simply being displayed on the screen, images can be used for texture maps, in which case they're essentially pasted onto polygons that are rendered on the screen in the normal way. (See Chapter 9 for more information about this technique.)

Reading, Writing, and Copying Pixel Data

OpenGL provides three basic commands that manipulate image data:

- **glReadPixels**() Reads a rectangular array of pixels from the framebuffer and stores the data in processor memory.
- **glDrawPixels**() Writes a rectangular array of pixels from data kept in processor memory into the framebuffer at the current raster position specified by **glRasterPos***().
- **glCopyPixels()** Copies a rectangular array of pixels from one part of the framebuffer to another. This command behaves similarly to a call to **glReadPixels()** followed by a call to **glDrawPixels()**, but the data is never written into processor memory.

For the aforementioned commands, the order of pixel data processing operations is shown in Figure 8-3:

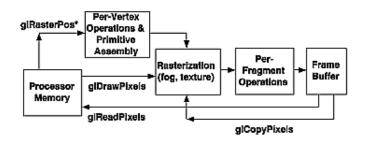


Figure 8-3 : Simplistic Diagram of Pixel Data Flow

The basic ideas in Figure 8-3 are correct. The coordinates of **glRasterPos***(), which specify the current raster position used by **glDrawPixels**() and **glCopyPixels**(), are transformed by the geometric processing pipeline. Both **glDrawPixels**() and **glCopyPixels**() are affected by rasterization and per-fragment operations. (But when drawing or copying a pixel rectangle, there's almost never a reason to have fog or texture enabled.)

However, additional steps arise because there are many kinds of framebuffer data, many ways to store pixel information in computer memory, and various data conversions that can be performed during the reading, writing, and copying operations. These possibilities translate to many different modes of operation. If all your program does is copy images on the screen or read them into memory temporarily so that they can be copied out later, you can ignore most of these modes. However, if you want your program to modify the data while it's in memory - for example, if you have an image stored in one format but the window requires a different format - or if you want to

save image data to a file for future restoration in another session or on another kind of machine with significantly different graphical capabilities, you have to understand the various modes.

The rest of this section describes the basic commands in detail. The following sections discuss the details of the series of imaging operations that comprise the Imaging Pipeline: pixel-storage modes, pixel-transfer operations, and pixel-mapping operations.

Reading Pixel Data from Frame Buffer to Processor Memory

void **glReadPixels**(*GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels*);

Reads pixel data from the framebuffer rectangle whose lower-left corner is at (x, y) and whose dimensions are width and height and stores it in the array pointed to by pixels. format indicates the kind of pixel data elements that are read (an index value or an R, G, B, or A component value, as listed in Table 8-1), and type indicates the data type of each element (see Table 8-2).

If you are using **glReadPixels**() to obtain RGBA or color-index information, you may need to clarify which buffer you are trying to access. For example, if you have a double-buffered window, you need to specify whether you are reading data from the front buffer or back buffer. To control the current read source buffer, call **glReadBuffer**(). (See "Selecting Color Buffers for Writing and Reading" in Chapter 10.)

Table 8-1 : Pixel Formats for glReadPixels() or glDrawPixels()

format Constant	Pixel Format
GL_COLOR_INDEX	A single color index
GL_RGB	A red color component, followed by a green color component, followed by a blue color component
GL_RGBA	A red color component, followed by a green color component, followed by a blue color component, followed by an alpha color component
GL_RED	A single red color component
GL_GREEN	A single green color component
GL_BLUE	A single blue color component
GL_ALPHA	A single alpha color component
GL_LUMINANCE	A single luminance component
GL_LUMINANCE_ALPHA	A luminance component followed by an alpha color component
GL_STENCIL_INDEX	A single stencil index
GL_DEPTH_COMPONENT	A single depth component

 Table 8-2 : Data Types for glReadPixels() or glDrawPixels()

type Constant	Data Type
GL_UNSIGNED_BYTE	unsigned 8-bit integer
GL_BYTE	signed 8-bit integer
GL_BITMAP	single bits in unsigned 8-bit integers using the same format as glBitmap()
GL_UNSIGNED_SHORT	unsigned 16-bit integer
GL_SHORT	signed 16-bit integer
GL_UNSIGNED_INT	unsigned 32-bit integer
GL_INT	signed 32-bit integer
GL_FLOAT	single-precision floating point

Remember that, depending on the format, anywhere from one to four elements are read (or written). For example, if the format is GL_RGBA and you're reading into 32-bit integers (that is, if *type* is equal to GL_UNSIGNED_INT or GL_INT), then every pixel read requires 16 bytes of storage (four components \times four bytes/component).

Each element of the image is stored in memory as indicated by Table 8-2. If the element represents a continuous value, such as a red, green, blue, or luminance component, each value is scaled to fit into the available number of bits. For example, assume the red component is initially specified as a floating-point value between 0.0 and 1.0. If it needs to be packed into an unsigned byte, only 8 bits of precision are kept, even if more bits are allocated to the red component in the framebuffer. GL_UNSIGNED_SHORT and GL_UNSIGNED_INT give 16 and 32 bits of precision, respectively. The normal (signed) versions of GL_BYTE, GL_SHORT, and GL_INT have 7, 15, and 31 bits of precision, since the negative values are typically not used.

If the element is an index (a color index or a stencil index, for example), and the type is not GL_FLOAT, the value is simply masked against the available bits in the type. The signed versions - GL_BYTE, GL_SHORT, and GL_INT - have masks with one fewer bit. For example, if a color index is to be stored in a signed 8-bit integer, it's first masked against 0x7f. If the type is GL_FLOAT, the index is simply converted into a single-precision floating-point number (for example, the index 17 is converted to the float 17.0).

Writing Pixel Data from Processor Memory to Frame Buffer

void glDrawPixels(GLsizei width, GLsizei height, GLenum format,

GLenum type, const GLvoid *pixels);

Draws a rectangle of pixel data with dimensions width and height. The pixel rectangle is drawn with its lower-left corner at the current raster position. format and type have the same meaning as with **glReadPixels()**. (For legal values for format and type, see Table 8-1 and

Table 8-2.) The array pointed to by pixels contains the pixel data to be drawn. If the current raster position is invalid, nothing is drawn, and the raster position remains invalid.

Example 8-3 is a portion of a program, which uses **glDrawPixels**() to draw an pixel rectangle in the lower-left corner of a window. **makeCheckImage**() creates a 64-by-64 RGB array of a black-and-white checkerboard image. **glRasterPos2i**(0,0) positions the lower-left corner of the image. For now, ignore **glPixelStorei**().

Example 8-3 : Use of glDrawPixels(): image.c

```
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageHeight][checkImageWidth][3];
void makeCheckImage(void)
ł
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {</pre>
      for (j = 0; j < checkImageWidth; j++)</pre>
         c = ((((i\&0x8)==0)^{((j\&0x8))==0}))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
      }
   }
}
void init(void)
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
void display(void)
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(0, 0);
   glDrawPixels(checkImageWidth, checkImageHeight, GL RGB,
                GL UNSIGNED BYTE, checkImage);
   glFlush();
}
```

When using **glDrawPixels**() to write RGBA or color-index information, you may need to control the current drawing buffers with **glDrawBuffer**(), which, along with **glReadBuffer**(), is also described in "Selecting Color Buffers for Writing and Reading" in Chapter 10.

Copying Pixel Data within the Frame Buffer

void **glCopyPixels**(*GLint x, GLint y, GLsizei width, GLsizei height, GLenum buffer*);

Copies pixel data from the framebuffer rectangle whose lower-left corner is at (x, y) and whose dimensions are width and height. The data is copied to a new position whose lower-left corner is given by the current raster position. buffer is either GL_COLOR, GL_STENCIL, or GL_DEPTH, specifying the framebuffer that is used. **glCopyPixels()** behaves similarly to a **glReadPixels()** followed by a **glDrawPixels()**, with the following translation for the buffer to format parameter:

- If *buffer* is GL_DEPTH or GL_STENCIL, then GL_DEPTH_COMPONENT or GL_STENCIL_INDEX is used, respectively.
- If GL_COLOR is specified, GL_RGBA or GL_COLOR_INDEX is used, depending on whether the system is in RGBA or color-index mode.

Note that there's no need for a *format* or *data* parameter for **glCopyPixels**(), since the data is never copied into processor memory. The read source buffer and the destination buffer of **glCopyPixels**() are specified by **glReadBuffer**() and **glDrawBuffer**() respectively. Both **glDrawPixels**() and **glCopyPixels**() are used in Example 8-4.

For all three functions, the exact conversions of the data going to or from the framebuffer depend on the modes in effect at the time. See the next section for details.

Imaging Pipeline

This section discusses the complete Imaging Pipeline: the pixel-storage modes and pixel-transfer operations, which include how to set up an arbitrary mapping to convert pixel data. You can also magnify or reduce a pixel rectangle before it's drawn by calling **glPixelZoom()**. The order of these operations is shown in Figure 8-4.

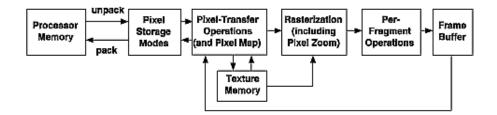


Figure 8-4 : Imaging Pipeline

When **glDrawPixels**() is called, the data is first unpacked from processor memory according to the pixel-storage modes that are in effect and then the pixel-transfer operations are applied. The resulting pixels are then rasterized. During rasterization, the pixel rectangle may be zoomed up or down, depending on the current state. Finally, the fragment operations are applied and the pixels are written into the framebuffer. (See "Testing and Operating on Fragments" in Chapter 10 for a discussion of the fragment operations.)

When **glReadPixels**() is called, data is read from the framebuffer, the pixel-transfer operations are performed, and then the resulting data is packed into processor memory.

glCopyPixels() applies all the pixel-transfer operations during what would be the **glReadPixels**() activity. The resulting data is written as it would be by **glDrawPixels**(), but the transformations aren't applied a second time. Figure 8-5 shows how **glCopyPixels**() moves pixel data, starting from the frame buffer.

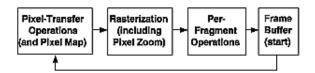


Figure 8-5 : glCopyPixels() Pixel Path

From "Drawing the Bitmap" and Figure 8-6, you see that rendering bitmaps is simpler than rendering images. Neither the pixel-transfer operations nor the pixel-zoom operation are applied.

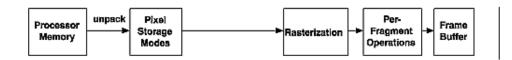


Figure 8-6 : glBitmap() Pixel Path

Note that the pixel-storage modes and pixel-transfer operations are applied to textures as they are read from or written to texture memory. Figure 8-7 shows the effect on **glTexImage***(), **glTexSubImage***(), and **glGetTexImage**().

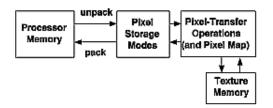


Figure 8-7 : glTexImage*(), glTexSubImage*(), and glGetTexImage() Pixel Paths

As seen in Figure 8-8, when pixel data is copied from the framebuffer into texture memory (glCopyTexImage*() or glCopyTexSubImage*()), only pixel-transfer operations are applied. (See Chapter 9 for more information on textures.)

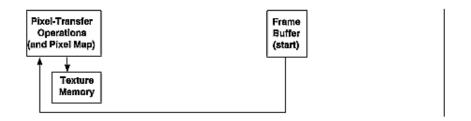


Figure 8-8 : glCopyTexImage*() and glCopyTexSubImage*() Pixel Paths

Pixel Packing and Unpacking

Packing and unpacking refer to the way that pixel data is written to and read from processor memory.

An image stored in memory has between one and four chunks of data, called *elements*. The data might consist of just the color index or the luminance (luminance is the weighted sum of the red, green, and blue values), or it might consist of the red, green, blue, and alpha components for each pixel. The possible arrangements of pixel data, or *formats*, determine the number of elements stored for each pixel and their order.

Some elements (such as a color index or a stencil index) are integers, and others (such as the red, green, blue, and alpha components, or the depth component) are floating-point values, typically ranging between 0.0 and 1.0. Floating-point components are usually stored in the framebuffer with lower resolution than a full floating-point number would require (for example, color components may be stored in 8 bits). The exact number of bits used to represent the component as a full 32-bit floating-point number, especially since images can easily contain a million pixels.

Elements can be stored in memory as various data types, ranging from 8-bit bytes to 32-bit integers or floating-point numbers. OpenGL explicitly defines the conversion of each component in each format to each of the possible data types. Keep in mind that you may lose data if you try to store a high-resolution component in a type represented by a small number of bits.

Controlling Pixel-Storage Modes

Image data is typically stored in processor memory in rectangular two- or three-dimensional arrays. Often, you want to display or store a subimage that corresponds to a subrectangle of the array. In addition, you might need to take into account that different machines have different byte-ordering conventions. Finally, some machines have hardware that is far more efficient at moving data to and from the framebuffer if the data is aligned on 2-byte, 4-byte, or 8-byte boundaries in processor memory. For such machines, you probably want to control the byte alignment. All the issues raised in this paragraph are controlled as pixel-storage modes, which are discussed in the next subsection. You specify these modes by using **glPixelStore***(), which you've already seen used in a couple of example programs.

All the possible pixel-storage modes are controlled with the **glPixelStore***() command. Typically, several successive calls are made with this command to set several parameter values.

void glPixelStore{if}(GLenum pname, TYPEparam);

Sets the pixel-storage modes, which affect the operation of glDrawPixels(), glReadPixels(), glBitmap(), glPolygonStipple(), glTexImage1D(), glTexImage2D(), glTexSubImage1D(), glTexSubImage2D(), and glGetTexImage(). The possible parameter names for pname are shown in Table 8-3, along with their data type, initial value, and valid range of values. The GL_UNPACK* parameters control how data is unpacked from memory by glDrawPixels(), glBitmap(), glPolygonStipple(), glTexImage1D(), glTexImage2D(), glTexSubImage1D(), and glTexSubImage2D(). The GL_PACK* parameters control how data is packed into memory by glReadPixels() and glGetTexImage().

Table 8-3 : glPixelStore() Parameters

Parameter Name	Туре	Initial Value	Valid Range
GL_UNPACK_SWAP_BYTES, GL_PACK_SWAP_BYTES	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST	GLboolean	FALSE	TRUE/FALSE
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	any nonnegative integer
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS	GLint	0	any nonnegative integer
GL_UNPACK_SKIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	any nonnegative integer
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT	GLint	4	1, 2, 4, 8

Since the corresponding parameters for packing and unpacking have the same meanings, they're discussed together in the rest of this section and referred to without the GL_PACK or GL_UNPACK prefix. For example, *SWAP_BYTES refers to GL_PACK_SWAP_BYTES and GL_UNPACK_SWAP_BYTES.

If the *SWAP_BYTES parameter is FALSE (the default), the ordering of the bytes in memory is whatever is native for the OpenGL client; otherwise, the bytes are reversed. The byte reversal applies to any size element, but really only has a meaningful effect for multibyte elements.

Note: As long as your OpenGL application doesn't share images with other machines, you can ignore the issue of byte ordering. If your application must render an OpenGL image that was created on a different machine and the "endianness" of the two machines differs, byte ordering can be swapped using *SWAP_BYTES. However, *SWAP_BYTES does not allow you to reorder elements (for example, to swap red and green).

The *LSB_FIRST parameter applies when drawing or reading 1-bit images or bitmaps, for which a single bit of data is saved or restored for each pixel. If *LSB_FIRST is FALSE (the default), the bits are taken from the bytes starting with the most significant bit; otherwise, they're taken in the opposite order. For example, if *LSB_FIRST is FALSE, and the byte in question is 0x31, the bits, in order, are $\{0, 0, 1, 1, 0, 0, 0, 1\}$. If *LSB_FIRST is TRUE, the order is $\{1, 0, 0, 0, 1, 1, 0, 0\}$.

Sometimes you want to draw or read only a subrectangle of the entire rectangle of image data stored in memory. If the rectangle in memory is larger than the subrectangle that's being drawn or read, you need to specify the actual length (measured in pixels) of the larger rectangle with *ROW_LENGTH. If *ROW_LENGTH is zero (which it is by default), the row length is understood to be the same as the width that's specified with **glReadPixels**(), **glDrawPixels**(), or

glCopyPixels(). You also need to specify the number of rows and pixels to skip before starting to copy the data for the subrectangle. These numbers are set using the parameters *SKIP_ROWS and *SKIP_PIXELS, as shown in Figure 8-9. By default, both parameters are 0, so you start at the lower-left corner.

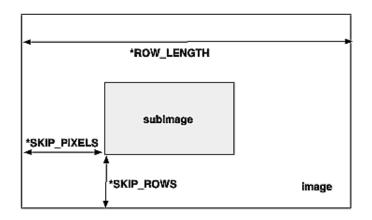


Figure 8-9 : *SKIP_ROWS, *SKIP_PIXELS, and *ROW_LENGTH Parameters

Often a particular machine's hardware is optimized for moving pixel data to and from memory, if the data is saved in memory with a particular byte alignment. For example, in a machine with 32-bit words, hardware can often retrieve data much faster if it's initially aligned on a 32-bit boundary, which typically has an address that is a multiple of 4. Likewise, 64-bit architectures might work better when the data is aligned to 8-byte boundaries. On some machines, however, byte alignment makes no difference.

As an example, suppose your machine works better with pixel data aligned to a 4-byte boundary. Images are most efficiently saved by forcing the data for each row of the image to begin on a 4-byte boundary. If the image is 5 pixels wide and each pixel consists of 1 byte each of red, green, and blue information, a row requires $5 \times 3 = 15$ bytes of data. Maximum display efficiency can be achieved if the first row, and each successive row, begins on a 4-byte boundary, so there is 1 byte of waste in the memory storage for each row. If your data is stored like this, set the *ALIGNMENT parameter appropriately (to 4, in this case).

If *ALIGNMENT is set to 1, the next available byte is used. If it's 2, a byte is skipped if necessary at the end of each row so that the first byte of the next row has an address that's a multiple of 2. In the case of bitmaps (or 1-bit images) where a single bit is saved for each pixel, the same byte alignment works, although you have to count individual bits. For example, if you're saving a single bit per pixel, the row length is 75, and the alignment is 4, then each row requires 75/8, or 9 3/8 bytes. Since 12 is the smallest multiple of 4 that is bigger than 9 3/8, 12 bytes of memory are used for each row. If the alignment is 1, then 10 bytes are used for each row, as 9 3/8 is rounded up to the next byte. (There is a simple use of **glPixelStorei**() in Example 8-4.)

Pixel-Transfer Operations

As image data is transferred from memory into the framebuffer, or from the framebuffer into memory, OpenGL can perform several operations on it. For example, the ranges of components can be altered - normally, the red component is between 0.0 and 1.0, but you might prefer to keep it in some other range; or perhaps the data you're using from a different graphics system stores the red

component in a different range. You can even create maps to perform arbitrary conversion of color indices or color components during pixel transfer. Conversions such as these performed during the transfer of pixels to and from the framebuffer are called pixel-transfer operations. They're controlled with the **glPixelTransfer*(**) and **glPixelMap*(**) commands.

Be aware that although the color, depth, and stencil buffers have many similarities, they don't behave identically, and a few of the modes have special cases for special buffers. All the mode details are covered in this section and the sections that follow, including all the special cases.

Some of the pixel-transfer function characteristics are set with **glPixelTransfer*()**. The other characteristics are specified with **glPixelMap*()**, which is described in the next section.

void glPixelTransfer{if}(GLenum pname, TYPEparam); Sets pixel-transfer modes that affect the operation of glDrawPixels(), glReadPixels(), glCopyPixels(), glTexImage1D(), glTexImage2D(), glCopyTexImage1D(), glCopyTexImage2D(), glTexSubImage1D(), glTexSubImage2D(), glCopyTexSubImage1D(), glCopyTexSubImage2D(), and glGetTexImage(). The parameter pname must be one of those listed in the first column of Table 8-4, and its value, param, must be in the valid range shown.

 Table 8-4 : glPixelTransfer*() Parameters (continued)

Parameter Name	Туре	Initial Value	Valid Range
GL_MAP_COLOR	GLboolean	FALSE	TRUE/FALSE
GL_MAP_STENCIL	GLboolean	FALSE	TRUE/FALSE
GL_INDEX_SHIFT	GLint	0	(- ∞ , ∞)
GL_INDEX_OFFSET	GLint	0	(- ∞ , ∞)
GL_RED_SCALE	GLfloat	1.0	(- ∞ , ∞)
GL_GREEN_SCALE	GLfloat	1.0	(- ∞ , ∞)
GL_BLUE_SCALE	GLfloat	1.0	(- ∞ , ∞)
GL_ALPHA_SCALE	GLfloat	1.0	(- ∞ , ∞)
GL_DEPTH_SCALE	GLfloat	1.0	(- ∞ , ∞)
GL_RED_BIAS	GLfloat	0	(- ∞ , ∞)
GL_GREEN_BIAS	GLfloat	0	(- ∞ , ∞)
GL_BLUE_BIAS	GLfloat	0	(- ∞ , ∞)
GL_ALPHA_BIAS	GLfloat	0	(- ∞ , ∞)
GL_DEPTH_BIAS	GLfloat	0	(- ∞ , ∞)

If the GL_MAP_COLOR or GL_MAP_STENCIL parameter is TRUE, then mapping is enabled. See the next subsection to learn how the mapping is done and how to change the contents of the maps. All the other parameters directly affect the pixel component values.

A scale and bias can be applied to the red, green, blue, alpha, and depth components. For example, you may wish to scale red, green, and blue components that were read from the framebuffer before converting them to a luminance format in processor memory. Luminance is computed as the sum of the red, green, and blue components, so if you use the default value for GL_RED_SCALE, GL_GREEN_SCALE and GL_BLUE_SCALE, the components all contribute equally to the final intensity or luminance value. If you want to convert RGB to luminance, according to the NTSC standard, you set GL_RED_SCALE to .30, GL_GREEN_SCALE to .59, and GL_BLUE_SCALE to .11.

Indices (color and stencil) can also be transformed. In the case of indices a shift and offset are applied. This is useful if you need to control which portion of the color table is used during

rendering.

Pixel Mapping

All the color components, color indices, and stencil indices can be modified by means of a table lookup before they are placed in screen memory. The command for controlling this mapping is **glPixelMap*()**.

void **glPixelMap**{ui us f}v(GLenum map, GLint mapsize,

const TYPE *values); Loads the pixel map indicated by map with mapsize entries, whose values are pointed to by

values. Table 8-5 lists the map names and values; the default sizes are all 1 and the default values are all 0. Each map's size must be a power of 2.

Map Name	Address	Value
GL_PIXEL_MAP_I_TO_I	color index	color index
GL_PIXEL_MAP_S_TO_S	stencil index	stencil index
GL_PIXEL_MAP_I_TO_R	color index	R
GL_PIXEL_MAP_I_TO_G	color index	G
GL_PIXEL_MAP_I_TO_B	color index	В
GL_PIXEL_MAP_I_TO_A	color index	А
GL_PIXEL_MAP_R_TO_R	R	R
GL_PIXEL_MAP_G_TO_G	G	G
GL_PIXEL_MAP_B_TO_B	В	В
GL_PIXEL_MAP_A_TO_A	А	А

 Table 8-5 : glPixelMap*() Parameter Names and Values

The maximum size of the maps is machine-dependent. You can find the sizes of the pixel maps supported on your machine with **glGetIntegerv()**. Use the query argument

GL_MAX_PIXEL_MAP_TABLE to obtain the maximum size for all the pixel map tables, and use GL_PIXEL_MAP_*_TO_*_SIZE to obtain the current size of the specified map. The six maps whose address is a color index or stencil index must always be sized to an integral power of 2. The four RGBA maps can be any size from 1 through GL_MAX_PIXEL_MAP_TABLE.

To understand how a table works, consider a simple example. Suppose that you want to create a

256-entry table that maps color indices to color indices using GL_PIXEL_MAP_I_TO_I. You create a table with an entry for each of the values between 0 and 255 and initialize the table with **glPixelMap***(). Assume you're using the table for thresholding and want to map indices below 101 (indices 0 to 100) to 0, and all indices 101 and above to 255. In this case, your table consists of 101 0s and 155 255s. The pixel map is enabled using the routine **glPixelTransfer***() to set the parameter GL_MAP_COLOR to TRUE. Once the pixel map is loaded and enabled, incoming color indices below 101 come out as 0, and incoming pixels between 101 and 255 are mapped to 255. If the incoming pixel is larger than 255, it's first masked by 255, throwing out all the bits above the eighth, and the resulting masked value is looked up in the table. If the incoming index is a floating-point value (say 88.14585), it's rounded to the nearest integer value (giving 88), and that number is looked up in the table (giving 0).

Using pixel maps, you can also map stencil indices or convert color indices to RGB. (See "Reading and Drawing Pixel Rectangles" for information about the conversion of indices.)

Magnifying, Reducing, or Flipping an Image

After the pixel-storage modes and pixel-transfer operations are applied, images and bitmaps are rasterized. Normally, each pixel in an image is written to a single pixel on the screen. However, you can arbitrarily magnify, reduce, or even flip (reflect) an image by using **glPixelZoom**().

void glPixelZoom(GLfloat zoomx, GLfloat zoomy);

Sets the magnification or reduction factors for pixel-write operations (*glDrawPixels(*) or *glCopyPixels(*)), in the x- and y-dimensions. By default, zoomx and zoomy are 1.0. If they're both 2.0, each image pixel is drawn to 4 screen pixels. Note that fractional magnification or reduction factors are allowed, as are negative factors. Negative zoom factors reflect the resulting image about the current raster position.

During rasterization, each image pixel is treated as a *zoomx* \times *zoomy* rectangle, and fragments are generated for all the pixels whose centers lie within the rectangle. More specifically, let (*x*rp, *y*rp) be the current raster position. If a particular group of elements (index or components) is the *n*th in a row and belongs to the *m*th column, consider the region in window coordinates bounded by the rectangle with corners at

(xrp + zoomx * n, yrp + zoomy * m) and (xrp + zoomx(n+1), yrp + zoomy(m+1))

Any fragments whose centers lie inside this rectangle (or on its bottom or left boundaries) are produced in correspondence with this particular group of elements.

A negative zoom can be useful for flipping an image. OpenGL describes images from the bottom row of pixels to the top (and from left to right). If you have a "top to bottom" image, such as a frame of video, you may want to use **glPixelZoom**(1.0, -1.0) to make the image right side up for OpenGL. Be sure that you reposition the current raster position appropriately, if needed.

Example 8-4 shows the use of **glPixelZoom**(). A checkerboard image is initially drawn in the lower-left corner of the window. Pressing a mouse button and moving the mouse uses **glCopyPixels**() to copy the lower-left corner of the window to the current cursor location. (If you copy the image onto itself, it looks wacky!) The copied image is zoomed, but initially it is zoomed by the default value of 1.0, so you won't notice. The 'z' and 'Z' keys increase and decrease the zoom factors by 0.5. Any window damage causes the contents of the window to be redrawn. Pressing the 'r' key resets the image and the zoom factors.

Example 8-4 : Drawing, Copying, and Zooming Pixel Data: image.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageHeight][checkImageWidth][3];
static GLdouble zoomFactor = 1.0;
static GLint height;
void makeCheckImage(void)
{
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {</pre>
      for (j = 0; j < checkImageWidth; j++)</pre>
         c = ((((i\&0x8)==0)^{((j\&0x8))==0}))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
      }
   }
}
void init(void)
ł
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(0, 0);
   glDrawPixels(checkImageWidth, checkImageHeight, GL_RGB,
                GL_UNSIGNED_BYTE, checkImage);
   glFlush();
}
void reshape(int w, int h)
ł
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   height = (GLint) h;
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, (GLdouble) w, 0.0, (GLdouble) h);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void motion(int x, int y)
ł
   static GLint screeny;
   screeny = height - (GLint) y;
   glRasterPos2i (x, screeny);
   glPixelZoom (zoomFactor, zoomFactor);
```

```
glCopyPixels (0, 0, checkImageWidth, checkImageHeight,
                 GL COLOR);
   glPixelZoom (1.0, 1.0);
   glFlush ();
}
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case `r':
      case 'R':
         zoomFactor = 1.0;
         glutPostRedisplay();
         printf ("zoomFactor reset to 1.0\n");
         break;
      case `z':
         zoomFactor += 0.5;
         if (zoomFactor >= 3.0)
            zoomFactor = 3.0;
         printf ("zoomFactor is now %4.1f\n", zoomFactor);
         break;
      case 'Z':
         zoomFactor -= 0.5;
         if (zoomFactor <= 0.5)
            zoomFactor = 0.5;
         printf ("zoomFactor is now %4.1f\n", zoomFactor);
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(250, 250);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMotionFunc(motion);
   glutMainLoop();
   return 0;
}
```

Reading and Drawing Pixel Rectangles

This section describes the reading and drawing processes in detail. The pixel conversions performed when going from framebuffer to memory (reading) are similar but not identical to the conversions performed when going in the opposite direction (drawing), as explained in the following sections. You may wish to skip this section the first time through, especially if you do not plan to use the pixel-transfer operations right away.

The Pixel Rectangle Drawing Process

Figure 8-10 and the following list describe the operation of drawing pixels into the framebuffer.

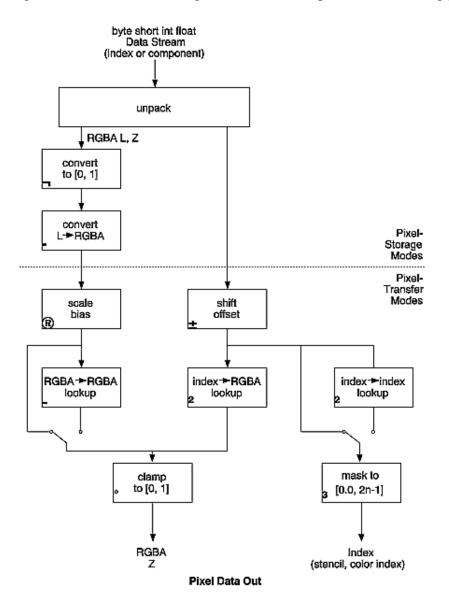


Figure 8-10 : Drawing Pixels with glDrawPixels()

- 1. If the pixels aren't indices (that is, the format isn't GL_COLOR_INDEX or GL_STENCIL_INDEX), the first step is to convert the components to floating-point format if necessary. (See Table 4-1 for the details of the conversion.)
- 2. If the format is GL_LUMINANCE or GL_LUMINANCE_ALPHA, the luminance element is converted into R, G, and B, by using the luminance value for each of the R, G, and B components. In GL_LUMINANCE_ALPHA format, the alpha value becomes the A value. If GL_LUMINANCE is specified, the A value is set to 1.0.
- 3. Each component (R, G, B, A, or depth) is multiplied by the appropriate scale, and the appropriate bias is added. For example, the R component is multiplied by the value

corresponding to GL_RED_SCALE and added to the value corresponding to GL_RED_BIAS.

- 4. If GL_MAP_COLOR is true, each of the R, G, B, and A components is clamped to the range [0.0,1.0], multiplied by an integer one less than the table size, truncated, and looked up in the table. (See "Tips for Improving Pixel Drawing Rates" for more details.)
- 5. Next, the R, G, B, and A components are clamped to [0.0,1.0], if they weren't already, and converted to fixed-point with as many bits to the left of the binary point as there are in the corresponding framebuffer component.
- 6. If you're working with index values (stencil or color indices), then the values are first converted to fixed-point (if they were initially floating-point numbers) with some unspecified bits to the right of the binary point. Indices that were initially fixed-point remain so, and any bits to the right of the binary point are set to zero.

The resulting index value is then shifted right or left by the absolute value of GL_INDEX_SHIFT bits; the value is shifted left if GL_INDEX_SHIFT > 0 and right otherwise. Finally, GL_INDEX_OFFSET is added to the index.

- 7. The next step with indices depends on whether you're using RGBA mode or color-index mode. In RGBA mode, a color index is converted to RGBA using the color components specified by GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B, and GL_PIXEL_MAP_I_TO_A. (See "Pixel Mapping" for details.) Otherwise, if GL_MAP_COLOR is GL_TRUE, a color index is looked up through the table GL_PIXEL_MAP_I_TO_I. (If GL_MAP_COLOR is GL_FALSE, the index is unchanged.) If the image is made up of stencil indices rather than color indices, and if GL_MAP_STENCIL is GL_TRUE, the index is looked up in the table corresponding to GL_PIXEL_MAP_S_TO_S. If GL_MAP_STENCIL is FALSE, the stencil index is unchanged.
- 8. Finally, if the indices haven't been converted to RGBA, the indices are then masked to the number of bits of either the color-index or stencil buffer, whichever is appropriate.

The Pixel Rectangle Reading Process

Many of the conversions done during the pixel rectangle drawing process are also done during the pixel rectangle reading process. The pixel reading process is shown in Figure 8-11 and described in the following list.

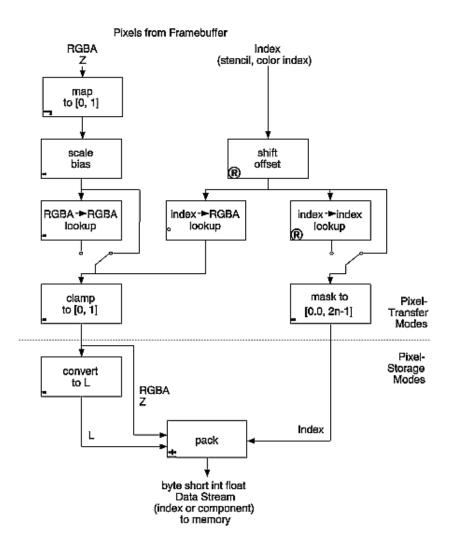


Figure 8-11 : Reading Pixels with glReadPixels()

- 1. If the pixels to be read aren't indices (that is, the format isn't GL_COLOR_INDEX or GL_STENCIL_INDEX), the components are mapped to [0.0,1.0] that is, in exactly the opposite way that they are when written.
- 2. Next, the scales and biases are applied to each component. If GL_MAP_COLOR is GL_TRUE, they're mapped and again clamped to [0.0,1.0]. If luminance is desired instead of RGB, the R, G, and B components are added (L = R + G + B).
- 3. If the pixels are indices (color or stencil), they're shifted, offset, and, if GL_MAP_COLOR is GL_TRUE, also mapped.
- 4. If the storage format is either GL_COLOR_INDEX or GL_STENCIL_INDEX, the pixel indices are masked to the number of bits of the storage type (1, 8, 16, or 32) and packed into memory as previously described.
- 5. If the storage format is one of the component kind (such as luminance or RGB), the pixels are always mapped by the index-to-RGBA maps. Then, they're treated as though they had been RGBA pixels in the first place (including potential conversion to luminance).

6. Finally, for both index and component data, the results are packed into memory according to the GL_PACK* modes set with **glPixelStore***().

The scaling, bias, shift, and offset values are the same as those used when drawing pixels, so if you're both reading and drawing pixels, be sure to reset these components to the appropriate values before doing a read or a draw. Similarly, the various maps must be properly reset if you intend to use maps for both reading and drawing.

Note: It might seem that luminance is handled incorrectly in both the reading and drawing operations. For example, luminance is not usually equally dependent on the R, G, and B components as it may be assumed from both Figure 8-10 and Figure 8-11. If you wanted your luminance to be calculated such that the R component contributed 30 percent, the G 59 percent, and the B 11 percent, you can set GL_RED_SCALE to .30, GL_RED_BIAS to 0.0, and so on. The computed L is then .30R + .59G + .11B.

Tips for Improving Pixel Drawing Rates

As you can see, OpenGL has a rich set of features for reading, drawing and manipulating pixel data. Although these features are often very useful, they can also decrease performance. Here are some tips for improving pixel draw rates.

- For best performance, set all pixel-transfer parameters to their default values, and set pixel zoom to (1.0,1.0).
- A series of fragment operations is applied to pixels as they are drawn into the framebuffer. (See "Testing and Operating on Fragments" in Chapter 10.) For optimum performance disable all fragment operations.
- While performing pixel operations, disable other costly states, such as texturing and lighting.
- If you use an image format and type that matches the framebuffer, you can reduce the amount of work that the OpenGL implementation has to do. For example, if you are writing images to an RGB framebuffer with 8 bits per component, call **glDrawPixels**() with *format* set to RGB and *type* set to UNSIGNED_BYTE.
- For some implementations, unsigned image formats are faster to use than signed image formats.
- It is usually faster to draw a large pixel rectangle than to draw several small ones, since the cost of transferring the pixel data can be amortized over many pixels.
- If possible, reduce the amount of data that needs to be copied by using small data types (for example, use GL_UNSIGNED_BYTE) and fewer components (for example, use format GL_LUMINANCE_ALPHA).
- Pixel-transfer operations, including pixel mapping and values for scale, bias, offset, and shift other than the defaults, may decrease performance.



OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 9 Texture Mapping

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand what texture mapping can add to your scene
- Specify a texture image
- Control how a texture image is filtered as it's applied to a fragment
- Create and manage texture images in texture objects and, if available, control a high-performance working set of those texture objects
- Specify how the color values in the image combine with those of the fragment to which it's being applied
- Supply texture coordinates to indicate how the texture image should be aligned to the objects in your scene
- Use automatic texture coordinate generation to produce effects like contour maps and environment maps

So far, every geometric primitive has been drawn as either a solid color or smoothly shaded between the colors at its vertices - that is, they've been drawn without texture mapping. If you want to draw a large brick wall without texture mapping, for example, each brick must be drawn as a separate polygon. Without texturing, a large flat wall - which is really a single rectangle - might require thousands of individual bricks, and even then the bricks may appear too smooth and regular to be realistic.

Texture mapping allows you to glue an image of a brick wall (obtained, perhaps, by scanning in a photograph of a real wall) to a polygon and to draw the entire wall as a single polygon. Texture mapping ensures that all the right things happen as the polygon is transformed and rendered. For example, when the wall is viewed in perspective, the bricks may appear smaller as the wall gets farther from the viewpoint. Other uses for texture mapping include depicting vegetation on large polygons representing the ground in flight simulation; wallpaper patterns; and textures that make polygons look like natural substances such as marble, wood, or cloth. The possibilities are endless. Although it's most natural to think of applying textures to polygons, textures can be applied to all primitives - points, lines, polygons, bitmaps, and images. Plates 6, 8, 18-21, 24-27, and 29-31 all demonstrate the use of textures.

Because there are so many possibilities, texture mapping is a fairly large, complex subject, and you

must make several programming choices when using it. For instance, you can map textures to surfaces made of a set of polygons or to curved surfaces, and you can repeat a texture in one or both directions to cover the surface. A texture can even be one-dimensional. In addition, you can automatically map a texture onto an object in such a way that the texture indicates contours or other properties of the item being viewed. Shiny objects can be textured so that they appear to be in the center of a room or other environment, reflecting the surroundings off their surfaces. Finally, a texture can be applied to a surface in different ways. It can be painted on directly (like a decal placed on a surface), used to modulate the color the surface would have been painted otherwise, or used to blend a texture color with the surface color. If this is your first exposure to texture mapping, you might find that the discussion in this chapter moves fairly quickly. As an additional reference, you might look at the chapter on texture mapping in *Fundamentals of Three-Dimensional Computer Graphics* by Alan Watt (Reading, MA: Addison-Wesley Publishing Company, 1990).

Textures are simply rectangular arrays of data - for example, color data, luminance data, or color and alpha data. The individual values in a texture array are often called *texels*. What makes texture mapping tricky is that a rectangular texture can be mapped to nonrectangular regions, and this must be done in a reasonable way.

Figure 9-1 illustrates the texture-mapping process. The left side of the figure represents the entire texture, and the black outline represents a quadrilateral shape whose corners are mapped to those spots on the texture. When the quadrilateral is displayed on the screen, it might be distorted by applying various transformations - rotations, translations, scaling, and projections. The right side of the figure shows how the texture-mapped quadrilateral might appear on your screen after these transformations. (Note that this quadrilateral is concave and might not be rendered correctly by OpenGL without prior tessellation. See Chapter 11 for more information about tessellating polygons.)

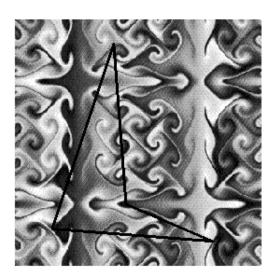




Figure 9-1 : Texture-Mapping Process

Notice how the texture is distorted to match the distortion of the quadrilateral. In this case, it's stretched in the x direction and compressed in the y direction; there's a bit of rotation and shearing going on as well. Depending on the texture size, the quadrilateral's distortion, and the size of the screen image, some of the texels might be mapped to more than one fragment, and some fragments

might be covered by multiple texels. Since the texture is made up of discrete texels (in this case, 256×256 of them), filtering operations must be performed to map texels to fragments. For example, if many texels correspond to a fragment, they're averaged down to fit; if texel boundaries fall across fragment boundaries, a weighted average of the applicable texels is performed. Because of these calculations, texturing is computationally expensive, which is why many specialized graphics systems include hardware support for texture mapping.

An application may establish texture objects, with each texture object representing a single texture (and possible associated mipmaps). Some implementations of OpenGL can support a special working set of texture objects that have better performance than texture objects outside the working set. These high-performance texture objects are said to be *resident* and may have special hardware and/or software acceleration available. You may use OpenGL to create and delete texture objects and to determine which textures constitute your working set.

This chapter covers the OpenGL's texture-mapping facility in the following major sections.

- "An Overview and an Example" gives a brief, broad look at the steps required to perform texture mapping. It also presents a relatively simple example of texture mapping.
- "Specifying the Texture" explains how to specify one- or two-dimensional textures. It also discusses how to use a texture's borders, how to supply a series of related textures of different sizes, and how to control the filtering methods used to determine how an applied texture is mapped to screen coordinates.
- "Filtering" details how textures are either magnified or minified as they are applied to the pixels of polygons. Minification using special mipmap textures is also explained.
- "Texture Objects" describes how to put texture images into objects so that you can control several textures at one time. With texture objects, you may be able to create a working set of high-performance textures, which are said to be resident. You may also prioritize texture objects to increase or decrease the likelihood that a texture object is resident.
- "Texture Functions" discusses the methods used for painting a texture onto a surface. You can choose to have the texture color values replace those that would be used if texturing wasn't in effect, or you can have the final color be a combination of the two.
- "Assigning Texture Coordinates" describes how to compute and assign appropriate texture coordinates to the vertices of an object. It also explains how to control the behavior of coordinates that lie outside the default range that is, how to repeat or clamp textures across a surface.
- "Automatic Texture-Coordinate Generation" shows how to have OpenGL automatically generate texture coordinates so that you can achieve such effects as contour and environment maps.
- "Advanced Features" explains how to manipulate the texture matrix stack and how to use the *q* texture coordinate.

Version 1.1 of OpenGL introduces several new texture-mapping operations:

- O Thirty-eight additional internal texture image formats
- O Texture proxy, to query whether there are enough resources to accommodate a given texture image
- Texture subimage, to replace all or part of an existing texture image rather than completely deleting and creating a texture to achieve the same effect
- O Specifying texture data from framebuffer memory (as well as from processor memory)
- O Texture objects, including resident textures and prioritizing

If you try to use one of these texture-mapping operations and can't find it, check the version number of your implementation of OpenGL to see if it actually supports it. (See "Which Version Am I Using?" in Chapter 14.)

An Overview and an Example

This section gives an overview of the steps necessary to perform texture mapping. It also presents a relatively simple texture-mapping program. Of course, you know that texture mapping can be a very involved process.

Steps in Texture Mapping

To use texture mapping, you perform these steps.

- 1. Create a texture object and specify a texture for that object.
- 2. Indicate how the texture is to be applied to each pixel.
- 3. Enable texture mapping.
- 4. Draw the scene, supplying both texture and geometric coordinates.

Keep in mind that texture mapping works only in RGBA mode. Texture mapping results in color-index mode are undefined.

Create a Texture Object and Specify a Texture for That Object

A texture is usually thought of as being two-dimensional, like most images, but it can also be one-dimensional. The data describing a texture may consist of one, two, three, or four elements per texel, representing anything from a modulation constant to an (R, G, B, A) quadruple.

In Example 9-1, which is very simple, a single texture object is created to maintain a single two-dimensional texture. This example does not find out how much memory is available. Since only one texture is created, there is no attempt to prioritize or otherwise manage a working set of texture objects. Other advanced techniques, such as texture borders or mipmaps, are not used in this simple example.

Indicate How the Texture Is to Be Applied to Each Pixel

You can choose any of four possible functions for computing the final RGBA value from the fragment color and the texture-image data. One possibility is simply to use the texture color as the final color; this is the *decal* mode, in which the texture is painted on top of the fragment, just as a decal would be applied. (Example 9-1 uses decal mode.) The *replace* mode, a variant of the decal mode, is a second method. Another method is to use the texture to *modulate*, or scale, the fragment's color; this technique is useful for combining the effects of lighting with texturing. Finally, a constant color can be blended with that of the fragment, based on the texture value.

Enable Texture Mapping

You need to enable texturing before drawing your scene. Texturing is enabled or disabled using **glEnable**() or **glDisable**() with the symbolic constant GL_TEXTURE_1D or GL_TEXTURE_2D for one- or two-dimensional texturing, respectively. (If both are enabled, GL_TEXTURE_2D is the one that is used.)

Draw the Scene, Supplying Both Texture and Geometric Coordinates

You need to indicate how the texture should be aligned relative to the fragments to which it's to be applied before it's "glued on." That is, you need to specify both texture coordinates and geometric coordinates as you specify the objects in your scene. For a two-dimensional texture map, for example, the texture coordinates range from 0.0 to 1.0 in both directions, but the coordinates of the items being textured can be anything. For the brick-wall example, if the wall is square and meant to represent one copy of the texture, the code would probably assign texture coordinates (0, 0), (1, 0), (1, 1), and (0, 1) to the four corners of the wall. If the wall is large, you might want to paint several copies of the texture map on it. If you do so, the texture map must be designed so that the bricks on the left edge match up nicely with the bricks on the right edge, and similarly for the bricks on the top and those on the bottom.

You must also indicate how texture coordinates outside the range [0.0,1.0] should be treated. Do the textures repeat to cover the object, or are they clamped to a boundary value?

A Sample Program

One of the problems with showing sample programs to illustrate texture mapping is that interesting textures are large. Typically, textures are read from an image file, since specifying a texture programmatically could take hundreds of lines of code. In Example 9-1, the texture - which consists of alternating white and black squares, like a checkerboard - is generated by the program. The program applies this texture to two squares, which are then rendered in perspective, one of them facing the viewer squarely and the other tilting back at 45 degrees, as shown in Figure 9-2. In object coordinates, both squares are the same size.

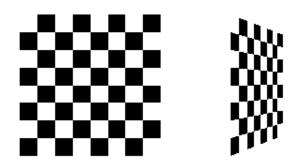


Figure 9-2 : Texture-Mapped Squares

Example 9-1 : Texture-Mapped Checkerboard: checker.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
/* Create checkerboard texture */
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLuint texName;
void makeCheckImage(void)
{
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {</pre>
      for (j = 0; j < checkImageWidth; j++)</pre>
         c = (((((i&0x8)==0)^((j&0x8))==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
      }
   }
}
void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glEnable(GL_DEPTH_TEST);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_2D, texName);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL NEAREST);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                checkImage);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glEnable(GL_TEXTURE_2D);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
  glBindTexture(GL_TEXTURE_2D, texName);
  glBegin(GL_QUADS);
  glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
  glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
  glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
  glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
  glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
  glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
  glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
  glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
  glEnd();
  glFlush();
  glDisable(GL TEXTURE 2D);
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 30.0);
  glMatrixMode(GL MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0, 0.0, -3.6);
}
void keyboard (unsigned char key, int x, int y)
ł
   switch (key) {
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(250, 250);
  glutInitWindowPosition(100, 100);
  glutCreateWindow(argv[0]);
   init();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

The checkerboard texture is generated in the routine **makeCheckImage**(), and all the texture-mapping initialization occurs in the routine **init**(). **glGenTextures**() and **glBindTexture**() name and create a texture object for a texture image. (See "Texture Objects.") The single, full-resolution texture map is specified by **glTexImage2D**(), whose parameters indicate the size of the image, type of the image, location of the image, and other properties of it. (See "Specifying the Texture" for more information about **glTexImage2D**().)

The four calls to **glTexParameter***() specify how the texture is to be wrapped and how the colors are to be filtered if there isn't an exact match between pixels in the texture and pixels on the screen. (See "Repeating and Clamping Textures" and "Filtering.")

In **display**(), **glEnable**() turns on texturing. **glTexEnv***() sets the drawing mode to GL_DECAL so that the textured polygons are drawn using the colors from the texture map (rather than taking into account what color the polygons would have been drawn without the texture).

Then, two polygons are drawn. Note that texture coordinates are specified along with vertex coordinates. The **glTexCoord***() command behaves similarly to the **glNormal**() command. **glTexCoord***() sets the current texture coordinates; any subsequent vertex command has those texture coordinates associated with it until **glTexCoord***() is called again.

Note: The checkerboard image on the tilted polygon might look wrong when you compile and run it on your machine - for example, it might look like two triangles with different projections of the checkerboard image on them. If so, try setting the parameter

GL_PERSPECTIVE_CORRECTION_HINT to GL_NICEST and running the example again. To do this, use **glHint**().

Specifying the Texture

The command **glTexImage2D**() defines a two-dimensional texture. It takes several arguments, which are described briefly here and in more detail in the subsections that follow. The related command for one-dimensional textures, **glTexImage1D**(), is described in "One-Dimensional Textures."

void **glTexImage2D**(GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels);

Defines a two-dimensional texture. The target parameter is set to either the constant GL_TEXTURE_2D or GL_PROXY_TEXTURE_2D. You use the level parameter if you're supplying multiple resolutions of the texture map; with only one resolution, level should be 0. (See "Multiple Levels of Detail" for more information about using multiple resolutions.) The next parameter, internalFormat, indicates which of the R, G, B, and A components or luminance or intensity values are selected for use in describing the texels of an image. The value of internalFormat is an integer from 1 to 4, or one of thirty-eight symbolic constants. The thirty-eight symbolic constants that are also legal values for internalFormat are GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE4, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA8, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, *GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, and GL_RGBA16. (See "Texture Functions" for a discussion of how these selected components are applied.)*

If internalFormat is one of the thirty-eight symbolic constants, then you are asking for specific components and perhaps the resolution of those components. For example, if internalFormat is GL_R3_G3_B2, you are asking that texels be 3 bits of red, 3 bits of green, and 2 bits of blue, but OpenGL is not guaranteed to deliver this. OpenGL is only obligated to choose an internal representation that closely approximates what is requested, but an exact match is usually not required. By definition, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, and GL_RGBA are lenient, because they do not ask for a specific resolution. (For compatibility with the OpenGL release 1.0, the numeric values 1, 2, 3, and 4, for internalFormat, are equivalent to the symbolic constants GL_LUMINANCE,

GL_LUMINANCE_ALPHA, GL_RGB, and GL_RGBA, respectively.)

The width and height parameters give the dimensions of the texture image; border indicates the width of the border, which is either zero (no border) or one. (See "Using a Texture's Borders".) Both width and height must have the form 2m+2b, where m is a nonnegative integer (which can have a different value for width than for height) and b is the value of border. The maximum size of a texture map depends on the implementation of OpenGL, but it must be at least 64×64 (or 66×66 with borders).

The format and type parameters describe the format and data type of the texture image data. They have the same meaning as they do for **glDrawPixels**(). (See "Imaging Pipeline" in Chapter 8.) In fact, texture data is in the same format as the data used by **glDrawPixels**(), so the settings of **glPixelStore***() and **glPixelTransfer***() are applied. (In Example 9-1, the call

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

is made because the data in the example isn't padded at the end of each texel row.) The format parameter can be GL_COLOR_INDEX, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, or GL_LUMINANCE_ALPHA that is, the same formats available for **glDrawPixels**() with the exceptions of GL_STENCIL_INDEX and GL_DEPTH_COMPONENT. Similarly, the type parameter can be GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, or GL_BITMAP. Finally, pixels contains the texture-image data. This data describes the texture image itself as well as its border.

The internal format of a texture image may affect the performance of texture operations. For example, some implementations perform texturing with GL_RGBA faster than GL_RGB, because the color components align the processor memory better. Since this varies, you should check specific information about your implementation of OpenGL.

The internal format of a texture image also may control how much memory a texture image consumes. For example, a texture of internal format GL_RGBA8 uses 32 bits per texel, while a texture of internal format GL_R3_G3_B2 only uses 8 bits per texel. Of course, there is a corresponding trade-off between memory consumption and color resolution.

Note: Although texture-mapping results in color-index mode are undefined, you can still specify a texture with a GL_COLOR_INDEX image. In that case, pixel-transfer operations are applied to

convert the indices to RGBA values by table lookup before they're used to form the texture image.

The number of texels for both the width and height of a texture image, not including the optional border, must be a power of 2. If your original image does not have dimensions that fit that limitation, you can use the OpenGL Utility Library routine **gluScaleImage()** to alter the size of your textures.

int **gluScaleImage**(*GLenum format, GLint widthin, GLint heightin, GLenum typein, const void *datain, GLint widthout, GLint heightout, GLenum typeout, void *dataout);*

Scales an image using the appropriate pixel-storage modes to unpack the data from datain. The format, typein, and typeout parameters can refer to any of the formats or data types supported by **glDrawPixels()**. The image is scaled using linear interpolation and box filtering (from the size indicated by widthin and heightin to widthout and heightout), and the resulting image is written to dataout, using the pixel GL_PACK* storage modes. The caller of **gluScaleImage()** must allocate sufficient space for the output buffer. A value of 0 is returned on success, and a GLU error code is returned on failure.

The framebuffer itself can also be used as a source for texture data. **glCopyTexImage2D**() reads a rectangle of pixels from the framebuffer and uses it for a new texture.

void glCopyTexImage2D(GLenum target, GLint level,

GLint internalFormat,

GLint x, GLint y, GLsizei width, GLsizei height,

GLint border);

Creates a two-dimensional texture, using framebuffer data to define the texels. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if glCopyPixels() had been called but stopped before final conversion. The settings of glPixelTransfer*() are applied.

The target parameter must be set to the constant GL_TEXTURE_2D. The level, internalFormat, and border parameters have the same effects that they have for **glTexImage2D()**. The texture array is taken from a screen-aligned pixel rectangle with the lower-left corner at coordinates specified by the (x, y) parameters. The width and height parameters specify the size of this pixel rectangle. Both width and height must have the form 2m+2b, where m is a nonnegative integer (which can have a different value for width than for height) and b is the value of border.

The next sections give more detail about texturing, including the use of the *target*, *border*, and *level* parameters. The *target* parameter can be used to accurately query the size of a texture (by creating a texture proxy with **glTexImage*D**()) and whether a texture possibly can be used within the texture resources of an OpenGL implementation. Redefining a portion of a texture is described in "Replacing All or Part of a Texture Image." One-dimensional textures are discussed in "One-Dimensional Textures." The texture border, which has its size controlled by the *border* parameter, is detailed in "Using a Texture's Borders." The *level* parameter is used to specify textures of different resolutions and is incorporated into the special technique of *mipmapping*, which is explained in "Multiple Levels of Detail." Mipmapping requires understanding how to filter textures as they're applied; filtering is the subject of "Filtering."

Texture Proxy

To an OpenGL programmer who uses textures, size is important. Texture resources are typically

limited and vary among OpenGL implementations. There is a special texture proxy target to evaluate whether sufficient resources are available.

glGetIntegerv(GL_MAX_TEXTURE_SIZE,...) tells you the largest dimension (width or height, without borders) of a texture image, typically the size of the largest square texture supported. However, GL_MAX_TEXTURE_SIZE does not consider the effect of the internal format of a texture. A texture image that stores texels using the GL_RGBA16 internal format may be using 64 bits per texel, so its image may have to be 16 times smaller than an image with the GL_LUMINANCE4 internal format. (Also, images requiring borders or mipmaps may further reduce the amount of available memory.)

A special place holder, or *proxy*, for a texture image allows the program to query more accurately whether OpenGL can accommodate a texture of a desired internal format. To use the proxy to query OpenGL, call **glTexImage2D**() with a *target* parameter of GL_PROXY_TEXTURE_2D and the given *level*, *internalFormat*, *width*, *height*, *border*, *format*, and *type*. (For one-dimensional textures, use corresponding 1D routines and symbolic constants.) For a proxy, you should pass NULL as the pointer for the *pixels* array.

To find out whether there are enough resources available for your texture, after the texture proxy has been created, query the texture state variables with **glGetTexLevelParameter***(). If there aren't enough resources to accommodate the texture proxy, the texture state variables for width, height, border width, and component resolutions are set to 0.

void glGetTexLevelParameter{if}v(GLenum target, GLint level,

GLenum pname, TYPE *params);

Returns in params texture parameter values for a specific level of detail, specified as level. target defines the target texture and is one of GL_TEXTURE_1D, GL_TEXTURE_2D, GL_PROXY_TEXTURE_1D, or GL_PROXY_TEXTURE_2D. Accepted values for pname are GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_BORDER, GL_TEXTURE_INTERNAL_FORMAT, GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_ALPHA_SIZE, GL_TEXTURE_LUMINANCE_SIZE, or GL_TEXTURE_INTENSITY_SIZE. GL_TEXTURE_COMPONENTS is also accepted for pname, but only for backward compatibility with OpenGL Release 1.0 - GL_TEXTURE_INTERNAL_FORMAT is the recommended symbolic constant for Release 1.1.

Example 9-2 demonstrates how to use the texture proxy to find out if there are enough resources to create a 64×64 texel texture with RGBA components with 8 bits of resolution. If this succeeds, then **glGetTexLevelParameteriv()** stores the internal format (in this case, GL_RGBA8) into the variable *format*.

Example 9-2 : Querying Texture Resources with a Texture Proxy

Note: There is one major limitation about texture proxies: The texture proxy tells you if there is space for your texture, but only if all texture resources are available (in other words, if it's the only texture in town). If other textures are using resources, then the texture proxy query may respond

affirmatively, but there may not be enough space to make your texture resident (that is, part of a possibly high-performance working set of textures). (See "Texture Objects" for more information about managing resident textures.)

Replacing All or Part of a Texture Image

Creating a texture may be more computationally expensive than modifying an existing one. In OpenGL Release 1.1, there are new routines to replace all or part of a texture image with new information. This can be helpful for certain applications, such as using real-time, captured video images as texture images. For that application, it makes sense to create a single texture and use **glTexSubImage2D**() to repeatedly replace the texture data with new video images. Also, there are no size restrictions for **glTexSubImage2D**() that force the height or width to be a power of two. This is helpful for processing video images, which generally do not have sizes that are powers of two.

void glTexSubImage2D(GLenum target, GLint level, GLint xoffset,

GLint yoffset, GLsizei width, GLsizei height,

GLenum format, GLenum type, const GLvoid *pixels);

Defines a two-dimensional texture image that replaces all or part of a contiguous subregion (in 2D, it's simply a rectangle) of the current, existing two-dimensional texture image. The target parameter must be set to GL_TEXTURE_2D.

The level, format, and type parameters are similar to the ones used for **glTexImage2D**(). level is the mipmap level-of-detail number. It is not an error to specify a width or height of zero, but the subimage will have no effect. format and type describe the format and data type of the texture image data. The subimage is also affected by modes set by **glPixelStore***() and **glPixelTransfer***().

pixels contains the texture data for the subimage. width and height are the dimensions of the subregion that is replacing all or part of the current texture image. xoffset and yoffset specify the texel offset in the x and y directions (with (0, 0) at the lower-left corner of the texture) and specify where to put the subimage within the existing texture array. This region may not include any texels outside the range of the originally defined texture array.

In Example 9-3, some of the code from Example 9-1 has been modified so that pressing the 's' key drops a smaller checkered subimage into the existing image. (The resulting texture is shown in Figure 9-3.) Pressing the 'r' key restores the original image. Example 9-3 shows the two routines, **makeCheckImages**() and **keyboard**(), that have been substantially changed. (See "Texture Objects" for more information about **glBindTexture**().)

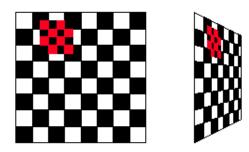


Figure 9-3 : Texture with Subimage Added

Example 9-3 : Replacing a Texture Subimage: texsub.c

```
/* Create checkerboard textures */
#define checkImageWidth 64
#define checkImageHeight 64
#define subImageWidth 16
#define subImageHeight 16
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLubyte subImage[subImageHeight][subImageWidth][4];
void makeCheckImages(void)
{
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {</pre>
      for (j = 0; j < checkImageWidth; j++)</pre>
         c = ((((i\&0x8)==0)^{(j\&0x8)})==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
      }
   for (i = 0; i < subImageHeight; i++) {
      for (j = 0; j < subImageWidth; j++)</pre>
         c = ((((i\&0x4)==0)^{(j\&0x4)})==0))*255;
         subImage[i][j][0] = (GLubyte) c;
         subImage[i][j][1] = (GLubyte) 0;
         subImage[i][j][2] = (GLubyte) 0;
         subImage[i][j][3] = (GLubyte) 255;
   }
}
void keyboard (unsigned char key, int x, int y)
   switch (key) {
      case `s':
      case `S':
         glBindTexture(GL_TEXTURE_2D, texName);
         glTexSubImage2D(GL_TEXTURE_2D, 0, 12, 44,
                          subImageWidth, subImageHeight, GL RGBA,
                          GL UNSIGNED BYTE, subImage);
         glutPostRedisplay();
         break;
      case `r':
      case `R':
         glBindTexture(GL_TEXTURE_2D, texName);
         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                       checkImageWidth, checkImageHeight, 0,
                       GL_RGBA, GL_UNSIGNED_BYTE, checkImage);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
```

Once again, the framebuffer itself can be used as a source for texture data; this time, a texture subimage. **glCopyTexSubImage2D**() reads a rectangle of pixels from the framebuffer and replaces a portion of an existing texture array. (**glCopyTexSubImage2D**() is kind of a cross between **glCopyTexImage2D**() and **glTexSubImage2D**().)

void glCopyTexSubImage2D(GLenum target, GLint level,

GLint xoffset, GLint yoffset, GLint x, GLint y, GLsizei width, GLsizei height);

Uses image data from the framebuffer to replace all or part of a contiguous subregion of the current, existing two-dimensional texture image. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if glCopyPixels() had been called, stopping before final conversion. The settings of glPixelStore*() and glPixelTransfer*() are applied.

The target parameter must be set to $GL_TEXTURE_2D$. level is the mipmap level-of-detail number. xoffset and yoffset specify the texel offset in the x and y directions (with (0, 0) at the lower-left corner of the texture) and specify where to put the subimage within the existing texture array. The subimage texture array is taken from a screen-aligned pixel rectangle with the lower-left corner at coordinates specified by the (x, y) parameters. The width and height parameters specify the size of this subimage rectangle.

One-Dimensional Textures

Sometimes a one-dimensional texture is sufficient - for example, if you're drawing textured bands where all the variation is in one direction. A one-dimensional texture behaves like a two-dimensional one with height = 1, and without borders along the top and bottom. All the two-dimensional texture and subtexture definition routines have corresponding one-dimensional routines. To create a simple one-dimensional texture, use **glTexImage1D**().

void glTexImage1D(GLenum target, GLint level, GLint internalFormat,

GLsizei width, GLint border, GLenum format,

GLenum type, const GLvoid *pixels);

Defines a one-dimensional texture. All the parameters have the same meanings as for glTexImage2D(), except that the image is now a one-dimensional array of texels. As before, the value of width is 2m (or 2m+2, if there's a border), where m is a nonnegative integer. You can supply mipmaps, proxies (set target to $GL_PROXY_TEXTURE_1D$), and the same filtering options are available as well.

For a sample program that uses a one-dimensional texture map, see Example 9-6.

To replace all or some of the texels of a one-dimensional texture, use glTexSubImage1D().

void glTexSubImage1D(GLenum target, GLint level, GLint xoffset,

GLsizei width, GLenum format,

GLenum type, const GLvoid *pixels);

Defines a one-dimensional texture array that replaces all or part of a contiguous subregion (in 1D, a row) of the current, existing one-dimensional texture image. The target parameter must be set to GL_TEXTURE_1D.

The level, format, and type parameters are similar to the ones used for **glTexImage1D**(). level is the mipmap level-of-detail number. format and type describe the format and data type of the texture image data. The subimage is also affected by modes set by **glPixelStore*()** or **glPixelTransfer*()**.

pixels contains the texture data for the subimage. *width* is the number of texels that replace part or all of the current texture image. *xoffset* specifies the texel offset for where to put the subimage within the existing texture array.

To use the framebuffer as the source of a new or replacement for an old one-dimensional texture, use either **glCopyTexImage1D()** or **glCopyTexSubImage1D()**.

void glCopyTexImage1D(GLenum target, GLint level,

GLint internalFormat, GLint x, GLint y,

GLsizei width, GLint border);

Creates a one-dimensional texture, using framebuffer data to define the texels. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if glCopyPixels() had been called but stopped before final conversion. The settings of glPixelStore*() and glPixelTransfer*() are applied.

The target parameter must be set to the constant $GL_TEXTURE_1D$. The level, internalFormat, and border parameters have the same effects that they have for glCopyTexImage2D(). The texture array is taken from a row of pixels with the lower-left corner at coordinates specified by the (x, y) parameters. The width parameter specifies the number of pixels in this row. The value of width is 2m (or 2m+2 if there's a border), where m is a nonnegative integer.

void **glCopyTexSubImage1D**(*GLenum target, GLint level, GLint xoffset, GLint x, GLint y, GLsizei width*);

Uses image data from the framebuffer to replace all or part of a contiguous subregion of the current, existing one-dimensional texture image. The pixels are read from the current GL_READ_BUFFER and are processed exactly as if glCopyPixels() had been called but stopped before final conversion. The settings of glPixelStore*() and glPixelTransfer*() are applied.

The target parameter must be set to $GL_TEXTURE_1D$. level is the mipmap level-of-detail number. xoffset specifies the texel offset and specifies where to put the subimage within the existing texture array. The subimage texture array is taken from a row of pixels with the lower-left corner at coordinates specified by the (x, y) parameters. The width parameter specifies the number of pixels in this row.

Using a Texture's Borders

Advanced

If you need to apply a larger texture map than your implementation of OpenGL allows, you can, with a little care, effectively make larger textures by tiling with several different textures. For example, if you need a texture twice as large as the maximum allowed size mapped to a square, draw the square as four subsquares, and load a different texture before drawing each piece.

Since only a single texture map is available at one time, this approach might lead to problems at the edges of the textures, especially if some form of linear filtering is enabled. The texture value to be used for pixels at the edges must be averaged with something beyond the edge, which, ideally, should come from the adjacent texture map. If you define a border for each texture whose texel values are equal to the values of the texels on the edge of the adjacent texture map, then the correct behavior results when linear filtering takes place.

To do this correctly, notice that each map can have eight neighbors - one adjacent to each edge, and one touching each corner. The values of the texels in the corner of the border need to correspond with the texels in the texture maps that touch the corners. If your texture is an edge or corner of the whole tiling, you need to decide what values would be reasonable to put in the borders. The easiest reasonable thing to do is to copy the value of the adjacent texel in the texture map. Remember that the border values need to be supplied at the same time as the texture-image data, so you need to figure this out ahead of time.

A texture's border color is also used if the texture is applied in such a way that it only partially covers a primitive. (See "Repeating and Clamping Textures" for more information about this situation.)

Multiple Levels of Detail

Advanced

Textured objects can be viewed, like any other objects in a scene, at different distances from the viewpoint. In a dynamic scene, as a textured object moves farther from the viewpoint, the texture map must decrease in size along with the size of the projected image. To accomplish this, OpenGL has to filter the texture map down to an appropriate size for mapping onto the object, without introducing visually disturbing artifacts. For example, to render a brick wall, you may use a large (say 128×128 texel) texture image when it is close to the viewer. But if the wall is moved farther away from the viewer until it appears on the screen as a single pixel, then the filtered textures may appear to change abruptly at certain transition points.

To avoid such artifacts, you can specify a series of prefiltered texture maps of decreasing resolutions, called *mipmaps*, as shown in Figure 9-4. The term *mipmap* was coined by Lance Williams, when he introduced the idea in his paper, "*Pyramidal Parametrics*" (SIGGRAPH 1983 Proceedings). *Mip* stands for the Latin *multim im parvo*, meaning "many things in a small place." Mipmapping uses some clever methods to pack image data into memory.

Original Texture

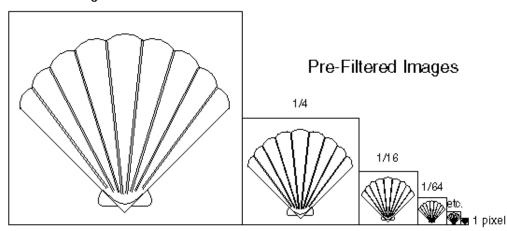


Figure 9-4 : Mipmaps

When using mipmapping, OpenGL automatically determines which texture map to use based on the size (in pixels) of the object being mapped. With this approach, the level of detail in the texture map is appropriate for the image that's drawn on the screen - as the image of the object gets smaller, the size of the texture map decreases. Mipmapping requires some extra computation and texture storage area; however, when it's not used, textures that are mapped onto smaller objects might shimmer and flash as the objects move.

To use mipmapping, you must provide all sizes of your texture in powers of 2 between the largest size and a 1×1 map. For example, if your highest-resolution map is 64×16 , you must also provide maps of size 32×8 , 16×4 , 8×2 , 4×1 , 2×1 , and 1×1 . The smaller maps are typically filtered and averaged-down versions of the largest map in which each texel in a smaller texture is an average of the corresponding four texels in the larger texture. (Since OpenGL doesn't require any particular method for calculating the smaller maps, the differently sized textures could be totally unrelated. In practice, unrelated textures would make the transitions between mipmaps extremely noticeable.)

To specify these textures, call **glTexImage2D**() once for each resolution of the texture map, with different values for the *level*, *width*, *height*, and *image* parameters. Starting with zero, *level* identifies which texture in the series is specified; with the previous example, the largest texture of size 64×16 would be declared with *level* = 0, the 32×8 texture with *level* = 1, and so on. In addition, for the mipmapped textures to take effect, you need to choose one of the appropriate filtering methods described in the next section.

Example 9-4 illustrates the use of a series of six texture maps decreasing in size from 32×32 to 1×1 . This program draws a rectangle that extends from the foreground far back in the distance, eventually disappearing at a point, as shown in "Plate 20" in Appendix I. Note that the texture coordinates range from 0.0 to 8.0 so 64 copies of the texture map are required to tile the rectangle, eight in each direction. To illustrate how one texture map succeeds another, each map has a different color.

Example 9-4 : Mipmap Textures: mipmap.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
GLubyte mipmapImage32[32][32][4];
GLubyte mipmapImage16[16][16][4];
GLubyte mipmapImage8[8][8][4];
GLubyte mipmapImage4[4][4][4];
GLubyte mipmapImage2[2][2][4];
GLubyte mipmapImage1[1][1][4];
static GLuint texName;
void makeImages(void)
{
   int i, j;
   for (i = 0; i < 32; i++) {
      for (j = 0; j < 32; j++) {
         mipmapImage32[i][j][0] = 255;
         mipmapImage32[i][j][1] = 255;
         mipmapImage32[i][j][2] = 0;
         mipmapImage32[i][j][3] = 255;
      }
   for (i = 0; i < 16; i++) {
      for (j = 0; j < 16; j++) {
         mipmapImage16[i][j][0] = 255;
         mipmapImage16[i][j][1] = 0;
         mipmapImage16[i][j][2] = 255;
         mipmapImage16[i][j][3] = 255;
      }
```

```
for (i = 0; i < 8; i++) {
      for (j = 0; j < 8; j++) {
         mipmapImage8[i][j][0] = 255;
         mipmapImage8[i][j][1] = 0;
         mipmapImage8[i][j][2] = 0;
         mipmapImage8[i][j][3] = 255;
      }
   for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++) {
         mipmapImage4[i][j][0] = 0;
         mipmapImage4[i][j][1] = 255;
         mipmapImage4[i][j][2] = 0;
         mipmapImage4[i][j][3] = 255;
      }
   for (i = 0; i < 2; i++) {
      for (j = 0; j < 2; j++) {
         mipmapImage2[i][j][0] = 0;
         mipmapImage2[i][j][1] = 0;
         mipmapImage2[i][j][2] = 255;
         mipmapImage2[i][j][3] = 255;
      }
   }
   mipmapImage1[0][0][0] = 255;
   mipmapImage1[0][0][1] = 255;
   mipmapImage1[0][0][2] = 255;
   mipmapImage1[0][0][3] = 255;
}
void init(void)
   glEnable(GL DEPTH TEST);
   glShadeModel(GL FLAT);
   glTranslatef(0.0, 0.0, -3.6);
   makeImages();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glGenTextures(1, &texName);
   glBindTexture(GL_TEXTURE_2D, texName);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST_MIPMAP_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage32);
   glTexImage2D(GL_TEXTURE_2D, 1, GL_RGBA, 16, 16, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage16);
   glTexImage2D(GL_TEXTURE_2D, 2, GL_RGBA, 8, 8, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage8);
   glTexImage2D(GL_TEXTURE_2D, 3, GL_RGBA, 4, 4, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage4);
   glTexImage2D(GL_TEXTURE_2D, 4, GL_RGBA, 2, 2, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage2);
   glTexImage2D(GL_TEXTURE_2D, 5, GL_RGBA, 1, 1, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, mipmapImage1);
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
   glEnable(GL_TEXTURE_2D);
}
```

```
void display(void)
   glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
  glBindTexture(GL TEXTURE 2D, texName);
  glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
   glTexCoord2f(0.0, 8.0); glVertex3f(-2.0, 1.0, 0.0);
   glTexCoord2f(8.0, 8.0); glVertex3f(2000.0, 1.0, -6000.0);
   glTexCoord2f(8.0, 0.0); glVertex3f(2000.0, -1.0, -6000.0);
   glEnd();
  glFlush();
}
void reshape(int w, int h)
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 30000.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
void keyboard (unsigned char key, int x, int y)
   switch (key) {
      case 27:
         exit(0);
         break;
     default:
         break;
   }
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(50, 50);
  glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

Example 9-4 illustrates mipmapping by making each mipmap a different color so that it's obvious when one map is replaced by another. In a real situation, you define mipmaps so that the transition is as smooth as possible. Thus, the maps of lower resolution are usually filtered versions of an original, high-resolution map. The construction of a series of such mipmaps is a software process, and thus isn't part of OpenGL, which is simply a rendering library. However, since mipmap construction is such an important operation, however, the OpenGL Utility Library contains two routines that aid in the manipulation of images to be used as mipmapped textures.

Assuming you have constructed the level 0, or highest-resolution map, the routines **gluBuild1DMipmaps**() and **gluBuild2DMipmaps**() construct and define the pyramid of mipmaps down to a resolution of 1×1 (or 1, for one-dimensional texture maps). If your original image has dimensions that are not exact powers of 2, **gluBuild*DMipmaps**() helpfully scales the image to the nearest power of 2.

int gluBuild1DMipmaps(GLenum target, GLint components, GLint width, GLenum format, GLenum type, void *data); int gluBuild2DMipmaps(GLenum target, GLint components, GLint width, GLint height, GLenum format, GLenum type, void *data); Constructs a series of mipmaps and calls glTexImage*D() to load the images. The parameters for target, components, width, height, format, type, and data are exactly

parameters for target, components, width, height, format, type, and data are exactly the same as those for **glTexImage1D()** and **glTexImage2D()**. A value of 0 is returned if all the mipmaps are constructed successfully; otherwise, a GLU error code is returned.

Filtering

Texture maps are square or rectangular, but after being mapped to a polygon or surface and transformed into screen coordinates, the individual texels of a texture rarely correspond to individual pixels of the final screen image. Depending on the transformations used and the texture mapping applied, a single pixel on the screen can correspond to anything from a tiny portion of a texel (magnification) to a large collection of texels (minification), as shown in Figure 9-5. In either case, it's unclear exactly which texel values should be used and how they should be averaged or interpolated. Consequently, OpenGL allows you to specify any of several filtering options to determine these calculations. The options provide different trade-offs between speed and image quality. Also, you can specify independently the filtering methods for magnification and minification.

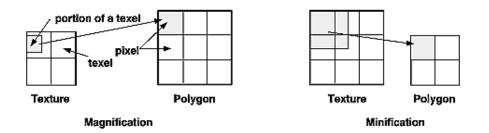


Figure 9-5 : Texture Magnification and Minification

In some cases, it isn't obvious whether magnification or minification is called for. If the mipmap needs to be stretched (or shrunk) in both the *x* and *y* directions, then magnification (or minification) is needed. If the mipmap needs to be stretched in one direction and shrunk in the other, OpenGL makes a choice between magnification and minification that in most cases gives the best result possible. It's best to try to avoid these situations by using texture coordinates that map without such distortion. (See "Computing Appropriate Texture Coordinates.")

The following lines are examples of how to use **glTexParameter***() to specify the magnification and minification filtering methods:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
```

The first argument to **glTexParameter***() is either GL_TEXTURE_2D or GL_TEXTURE_1D, depending on whether you're working with two- or one-dimensional textures. For the purposes of this discussion, the second argument is either GL_TEXTURE_MAG_FILTER or GL_TEXTURE_MIN_FILTER to indicate whether you're specifying the filtering method for magnification or minification. The third argument specifies the filtering method; Table 9-1 lists the possible values.

Parameter	Values
GL_TEXTURE_MAG_FILTER	GL_NEAREST or GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, or GL_LINEAR_MIPMAP_LINEAR

If you choose GL_NEAREST, the texel with coordinates nearest the center of the pixel is used for both magnification and minification. This can result in aliasing artifacts (sometimes severe). If you choose GL_LINEAR, a weighted linear average of the 2×2 array of texels that lie nearest to the center of the pixel is used, again for both magnification and minification. When the texture coordinates are near the edge of the texture map, the nearest 2×2 array of texels might include some that are outside the texture map. In these cases, the texel values used depend on whether GL_REPEAT or GL_CLAMP is in effect and whether you've assigned a border for the texture. (See "Using a Texture's Borders.") GL_NEAREST requires less computation than GL_LINEAR and therefore might execute more quickly, but GL_LINEAR provides smoother results.

With magnification, even if you've supplied mipmaps, the largest texture map (level = 0) is always used. With minification, you can choose a filtering method that uses the most appropriate one or two mipmaps, as described in the next paragraph. (If GL_NEAREST or GL_LINEAR is specified with minification, the largest texture map is used.)

As shown in Table 9-1, four additional filtering choices are available when minifying with mipmaps. Within an individual mipmap, you can choose the nearest texel value with GL_NEAREST_MIPMAP_NEAREST, or you can interpolate linearly by specifying GL_LINEAR_MIPMAP_NEAREST. Using the nearest texels is faster but yields less desirable results. The particular mipmap chosen is a function of the amount of minification required, and there's a cutoff point from the use of one particular mipmap to the next. To avoid a sudden transition, use GL_NEAREST_MIPMAP_LINEAR or GL_LINEAR_MIPMAP_LINEAR to linearly interpolate texel values from the two nearest best choices of mipmaps. GL_NEAREST_MIPMAP_LINEAR selects the nearest texel in each of the two maps and then interpolates linearly between these two values. GL_LINEAR_MIPMAP_LINEAR uses linear interpolation to compute the value in each of two maps and then interpolates linearly between these two values. As you might expect, GL_LINEAR_MIPMAP_LINEAR generally produces the smoothest results, but it requires the most computation and therefore might be the slowest.

Texture Objects

Texture objects are an important new feature in release 1.1 of OpenGL. A texture object stores texture data and makes it readily available. You can now control many textures and go back to textures that have been previously loaded into your texture resources. Using texture objects is usually the fastest way to apply textures, resulting in big performance gains, because it is almost always much faster to bind (reuse) an existing texture object than it is to reload a texture image using **glTexImage*D**().

Also, some implementations support a limited working set of high-performance textures. You can use texture objects to load your most often used textures into this limited area.

To use texture objects for your texture data, take these steps.

- 1. Generate texture names.
- 2. Initially bind (create) texture objects to texture data, including the image arrays and texture properties.
- 3. If your implementation supports a working set of high-performance textures, see if you have enough space for all your texture objects. If there isn't enough space, you may wish to establish priorities for each texture object so that more often used textures stay in the working set.
- 4. Bind and rebind texture objects, making their data currently available for rendering textured models.

Naming A Texture Object

Any nonzero unsigned integer may be used as a texture name. To avoid accidentally reusing names, consistently use **glGenTextures**() to provide unused texture names.

void glGenTextures(GLsizei n, GLuint *textureNames);

Returns n currently unused names for texture objects in the array textureNames. The names returned in textureNames do not have to be a contiguous set of integers. The names in textureNames are marked as used, but they acquire texture state and dimensionality (1D or 2D) only when they are first bound. Zero is a reserved texture name and is never returned as a texture name by **glGenTextures**().

glIsTexture() determines if a texture name is actually in use. If a texture name was returned by **glGenTextures**() but has not yet been bound (calling **glBindTexture**() with the name at least once), then **glIsTexture**() returns GL_FALSE.

GLboolean gllsTexture(GLuint textureName);

Returns GL_TRUE if textureName is the name of a texture that has been bound and has not been subsequently deleted. Returns GL_FALSE if textureName is zero or textureName is a nonzero value that is not the name of an existing texture.

Creating and Using Texture Objects

The same routine, **glBindTexture**(), both creates and uses texture objects. When a texture name is initially bound (used with **glBindTexture**()), a new texture object is created with default values for the texture image and texture properties. Subsequent calls to **glTexImage***(), **glTexSubImage***(), **glCopyTexSubImage***(), **glTexSubImage***(), **glCopyTexSubImage***(), **glTexParameter***(), and **glPrioritizeTextures**() store data in the texture object. The texture object may contain a texture image and associated mipmap images (if any), including associated data such as width, height, border width, internal format, resolution of components, and texture properties. Saved texture properties include minification filters, wrapping modes, border color, and texture priority.

When a texture object is subsequently bound once again, its data becomes the current texture state. (The state of the previously bound texture is replaced.)

void glBindTexture(GLenum target, GLuint textureName);

glBindTexture() does three things. When using textureName of an unsigned integer other than zero for the first time, a new texture object is created and assigned that name. When binding to a previously created texture object, that texture object becomes active. When binding to a textureName value of zero, OpenGL stops using texture objects and returns to the unnamed default texture.

When a texture object is initially bound (that is, created), it assumes the dimensionality of target, which is either GL_TEXTURE_1D or GL_TEXTURE_2D. Immediately upon its initial binding, the state of texture object is equivalent to the state of the default GL_TEXTURE_1D or GL_TEXTURE_2D (depending upon its dimensionality) at the initialization of OpenGL. In this initial state, texture properties such as minification and magnification filters, wrapping modes, border color, and texture priority are set to their default values.

In Example 9-5, two texture objects are created in **init()**. In **display()**, each texture object is used to render a different four-sided polygon.

Example 9-5 : Binding Texture Objects: texbind.c

```
#define checkImageWidth 64
#define checkImageHeight 64
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];
static GLubyte otherImage[checkImageHeight][checkImageWidth][4];
static GLuint texName[2];
void makeCheckImages(void)
   int i, j, c;
   for (i = 0; i < checkImageHeight; i++) {</pre>
      for (j = 0; j < checkImageWidth; j++)</pre>
         c = (((((i&0x8)==0)^((j&0x8))==0))*255;
         checkImage[i][j][0] = (GLubyte) c;
         checkImage[i][j][1] = (GLubyte) c;
         checkImage[i][j][2] = (GLubyte) c;
         checkImage[i][j][3] = (GLubyte) 255;
         c = (((((i&0x10)==0)^((j&0x10))==0))*255;
         otherImage[i][j][0] = (GLubyte) c;
         otherImage[i][j][1] = (GLubyte) 0;
         otherImage[i][j][2] = (GLubyte) 0;
         otherImage[i][j][3] = (GLubyte) 255;
      }
   }
}
```

```
void init(void)
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL FLAT);
  glEnable(GL_DEPTH_TEST);
   makeCheckImages();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glGenTextures(2, texName);
   glBindTexture(GL TEXTURE 2D, texName[0]);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL NEAREST);
   glTexParameteri(GL TEXTURE 2D, GL TEXTURE MIN FILTER,
                   GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                checkImage);
   glBindTexture(GL TEXTURE 2D, texName[1]);
   glTexParameteri(GL TEXTURE 2D, GL TEXTURE WRAP S, GL CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexParameteri(GL TEXTURE 2D, GL TEXTURE MAG FILTER,
                   GL NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL NEAREST);
   glTexEnvf(GL TEXTURE ENV, GL TEXTURE ENV MODE, GL DECAL);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, checkImageWidth,
                checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                otherImage);
   glEnable(GL TEXTURE 2D);
}
void display(void)
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBindTexture(GL_TEXTURE_2D, texName[0]);
   glBegin(GL_QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(-2.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(0.0, 1.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
   glEnd();
   glBindTexture(GL TEXTURE 2D, texName[1]);
   glBegin(GL QUADS);
   glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
   glTexCoord2f(0.0, 1.0); glVertex3f(1.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(2.41421, 1.0, -1.41421);
   glTexCoord2f(1.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
  glEnd();
   glFlush();
}
```

Whenever a texture object is bound once again, you may edit the contents of the bound texture object. Any commands you call that change the texture image or other properties change the contents of the currently bound texture object as well as the current texture state.

In Example 9-5, after completion of **display**(), you are still bound to the texture named by the contents of *texName*[1]. Be careful that you don't call a spurious texture routine that changes the data in that texture object.

When using mipmaps, all related mipmaps of a single texture image must be put into a single texture object. In Example 9-4, levels 0-5 of a mipmapped texture image are put into a single texture object named *texName*.

Cleaning Up Texture Objects

As you bind and unbind texture objects, their data still sits around somewhere among your texture resources. If texture resources are limited, deleting textures may be one way to free up resources.

void glDeleteTextures(GLsizei n, const GLuint *textureNames);

Deletes n texture objects, named by elements in the array textureNames. The freed texture names may now be reused (for example, by **glGenTextures**()). If a texture that is currently bound is deleted, the binding reverts to the default texture, as if **glBindTexture**() were called with zero for the value of textureName. Attempts to delete nonexistent texture names or the texture name of zero are ignored without generating an error.

A Working Set of Resident Textures

Some OpenGL implementations support a working set of high-performance textures, which are said to be resident. Typically, these implementations have specialized hardware to perform texture operations and a limited hardware cache to store texture images. In this case, using texture objects is recommended, because you are able to load many textures into the working set and then control them.

If all the textures required by the application exceed the size of the cache, some textures cannot be resident. If you want to find out if a single texture is currently resident, bind its object, and then use **glGetTexParameter*v()** to find out the value associated with the GL_TEXTURE_RESIDENT state. If you want to know about the texture residence status of many textures, use **glAreTexturesResident()**.

GLboolean **glAreTexturesResident**(GLsizei n, const GLuint*textureNames, GLboolean *residences);

Queries the texture residence status of the n texture objects, named in the array textureNames. residences is an array in which texture residence status is returned for the corresponding texture objects in the array textureNames. If all the named textures in textureNames are resident, the **glAreTexturesResident()** function returns GL_TRUE, and the contents of the array residences are undisturbed. If any texture in textureNames is not resident, then **glAreTexturesResident()** returns GL_FALSE and the elements in residences, which correspond to nonresident texture objects in textureNames, are also set to GL_FALSE.

Note that **glAreTexturesResident()** returns the current residence status. Texture resources are very dynamic, and texture residence status may change at any time. Some implementations cache textures when they are first used. It may be necessary to draw with the texture before checking residency.

If your OpenGL implementation does not establish a working set of high-performance textures, then the texture objects are always considered resident. In that case, **glAreTexturesResident**() always returns GL_TRUE and basically provides no information.

Texture Residence Strategies

If you can create a working set of textures and want to get the best texture performance possible, you really have to know the specifics of your implementation and application. For example, with a visual simulation or video game, you have to maintain performance in all situations. In that case, you should never access a nonresident texture. For these applications, you want to load up all your textures upon initialization and make them all resident. If you don't have enough texture memory available, you may need to reduce the size, resolution, and levels of mipmaps for your texture images, or you may use **glTexSubImage***() to repeatedly reuse the same texture memory.

For applications that create textures "on the fly," nonresident textures may be unavoidable. If some textures are used more frequently than others, you may assign a higher priority to those texture objects to increase their likelihood of being resident. Deleting texture objects also frees up space. Short of that, assigning a lower priority to a texture object may make it first in line for being moved out of the working set, as resources dwindle. **glPrioritizeTextures**() is used to assign priorities to texture objects.

void glPrioritizeTextures(GLsizei n, const GLuint *textureNames,

const GLclampf *priorities);

Assigns the n texture objects, named in the array textureNames, the texture residence priorities in the corresponding elements of the array priorities. The priority values in the array priorities are clamped to the range [0.0, 1.0] before being assigned. Zero indicates the lowest priority; these textures are least likely to be resident. One indicates the highest priority.

glPrioritizeTextures() does not require that any of the textures in textureNames be bound. However, the priority might not have any effect on a texture object until it is initially bound.

glTexParameter*() also may be used to set a single texture's priority, but only if the texture is currently bound. In fact, use of **glTexParameter***() is the only way to set the priority of a default texture.

If texture objects have equal priority, typical implementations of OpenGL apply a least recently used (LRU) strategy to decide which texture objects to move out of the working set. If you know that your OpenGL implementation has this behavior, then having equal priorities for all texture objects creates a reasonable LRU system for reallocating texture resources.

If your implementation of OpenGL doesn't use an LRU strategy for texture objects of equal priority (or if you don't know how it decides), you can implement your own LRU strategy by carefully maintaining the texture object priorities. When a texture is used (bound), you can maximize its priority, which reflects its recent use. Then, at regular (time) intervals, you can degrade the priorities of all texture objects.

Note: Fragmentation of texture memory can be a problem, especially if you're deleting and creating lots of new textures. Although it is even possible that you can load all the texture objects into a working set by binding them in one sequence, binding them in a different sequence may leave some textures nonresident.

Texture Functions

In all the examples so far in this chapter, the values in the texture map have been used directly as colors to be painted on the surface being rendered. You can also use the values in the texture map to

modulate the color that the surface would be rendered without texturing, or to blend the color in the texture map with the original color of the surface. You choose one of four texturing functions by supplying the appropriate arguments to **glTexEnv***().

void glTexEnv{if}(GLenum target, GLenum pname, TYPEparam);

void **glTexEnv**{*if*}**v**(*GLenum target, GLenum pname, TYPE *param*); Sets the current texturing function. target must be *GL_TEXTURE_ENV*. If pname is *GL_TEXTURE_ENV_MODE, param can be GL_DECAL, GL_REPLACE, GL_MODULATE,* or *GL_BLEND, to specify how texture values are to be combined with the color values of the* fragment being processed. If pname is *GL_TEXTURE_ENV_COLOR, param is an array of* four floating-point values representing *R, G, B, and A components. These values are used* only if the *GL_BLEND texture function has been specified as well.*

The combination of the texturing function and the base internal format determine how the textures are applied for each component of the texture. The texturing function operates on selected components of the texture and the color values that would be used with no texturing. (Note that the selection is performed after the pixel-transfer function has been applied.) Recall that when you specify your texture map with **glTexImage*D**(), the third argument is the internal format to be selected for each texel.

Table 9-2 and Table 9-3 show how the texturing function and base internal format determine the texturing application formula used for each component of the texture. There are six base internal formats (the letters in parentheses represent their values in the tables): GL_ALPHA (A), GL_LUMINANCE (L), GL_LUMINANCE_ALPHA (L and A), GL_INTENSITY (I), GL_RGB (C), and GL_RGBA (C and A). Other internal formats specify desired resolutions of the texture components and can be matched to one of these six base internal formats.

Base Internal Format	Replace Texture Function	Modulate Texture Function
GL_ALPHA	C = Cf, A = At	C = Cf, A = AfAt
GL_LUMINANCE	C = Lt, A = Af	C = CfLt, A = Af
GL_LUMINANCE_ALPHA	C = Lt, A = At	C = CfLt, A = AfAt
GL_INTENSITY	C = It, A = It	C = CfIt, A = AfIt
GL_RGB	C = Ct, A = Af	C = CfCt, A = Af
GL_RGBA	C = Ct, A = At	C = CfCt, A = AfAt

 Table 9-2 : Replace and Modulate Texture Function

Base Internal Format	Decal Texture Function	Blend Texture Function
GL_ALPHA	undefined	C = Cf, A = AfAt
GL_LUMINANCE	undefined	C = Cf(1-Lt) + CcLt, A = Af
GL_LUMINANCE_ALPHA	undefined	C = Cf(1-Lt) + CcLt, A = AfAt
GL_INTENSITY	undefined	C = Cf(1-It) + CcIt, A = Af(1-It) + AcIt,
GL_RGB	C = Ct, A = Af	C = Cf(1-Ct) + CcCt, A = Af
GL_RGBA	C = Cf(1-At) + CtAt, A = Af	C = Cf(1-Ct) + CcCt, A = AfAt

 Table 9-3 : Decal and Blend Texture Function

Note: In Table 9-2 and Table 9-3, a subscript of t indicates a texture value, f indicates the incoming fragment value, c indicates the values assigned with GL_TEXTURE_ENV_COLOR, and no subscript indicates the final, computed value. Also in the tables, multiplication of a color triple by a scalar means multiplying each of the R, G, and B components by the scalar; multiplying (or adding) two color triples means multiplying (or adding) each component of the second by the corresponding component of the first.

The decal texture function makes sense only for the RGB and RGBA internal formats (remember that texture mapping doesn't work in color-index mode). With the RGB internal format, the color that would have been painted in the absence of any texture mapping (the fragment's color) is replaced by the texture color, and its alpha is unchanged. With the RGBA internal format, the fragment's color is blended with the texture color in a ratio determined by the texture alpha, and the fragment's alpha is unchanged. You use the decal texture function in situations where you want to apply an opaque texture to an object - if you were drawing a soup can with an opaque label, for example. The decal texture function also can be used to apply an alpha blended texture, such as an insignia onto an airplane wing.

The replacement texture function is similar to decal; in fact, for the RGB internal format, they are exactly the same. With all the internal formats, the component values are either replaced or left alone.

For modulation, the fragment's color is modulated by the contents of the texture map. If the base internal format is GL_LUMINANCE, GL_LUMINANCE_ALPHA, or GL_INTENSITY, the color

values are multiplied by the same value, so the texture map modulates between the fragment's color (if the luminance or intensity is 1) to black (if it's 0). For the GL_RGB and GL_RGBA internal formats, each of the incoming color components is multiplied by a corresponding (possibly different) value in the texture. If there's an alpha value, it's multiplied by the fragment's alpha. Modulation is a good texture function for use with lighting, since the lit polygon color can be used to attenuate the texture color. Most of the texture-mapping examples in the color plates use modulation for this reason. White, specular polygons are often used to render lit, textured objects, and the texture image provides the diffuse color.

The blending texture function is the only function that uses the color specified by GL_TEXTURE_ENV_COLOR. The luminance, intensity, or color value is used somewhat like an alpha value to blend the fragment's color with the GL_TEXTURE_ENV_COLOR. (See "Sample Uses of Blending" in Chapter 6 for the billboarding example, which uses a blended texture.)

Assigning Texture Coordinates

As you draw your texture-mapped scene, you must provide both object coordinates and texture coordinates for each vertex. After transformation, the object coordinates determine where on the screen that particular vertex is rendered. The texture coordinates determine which texel in the texture map is assigned to that vertex. In exactly the same way that colors are interpolated between two vertices of shaded polygons and lines, texture coordinates are also interpolated between vertices. (Remember that textures are rectangular arrays of data.)

Texture coordinates can comprise one, two, three, or four coordinates. They're usually referred to as the *s*, *t*, *r*, and *q* coordinates to distinguish them from object coordinates (*x*, *y*, *z*, and *w*) and from evaluator coordinates (*u* and *v*; see Chapter 12). For one-dimensional textures, you use the *s* coordinate; for two-dimensional textures, you use *s* and *t*. In Release 1.1, the *r* coordinate is ignored. (Some implementations have 3D texture mapping as an extension, and that extension uses the *r* coordinate.) The *q* coordinate, like *w*, is typically given the value 1 and can be used to create homogeneous coordinates; it's described as an advanced feature in "The q Coordinate." The command to specify texture coordinates, **glTexCoord***(), is similar to **glVertex***(), **glColor***(), and **glNormal***() - it comes in similar variations and is used the same way between **glBegin**() and **glEnd**() pairs. Usually, texture-coordinate values range from 0 to 1; values can be assigned outside this range, however, with the results described in "Repeating and Clamping Textures."

void glTexCoord{1234}{sifd}(TYPEcoords);

void glTexCoord{1234}{sifd}v(TYPE *coords);

Sets the current texture coordinates (s, t, r, q). Subsequent calls to $glVertex^*()$ result in those vertices being assigned the current texture coordinates. With $glTexCoord1^*()$, the s coordinate is set to the specified value, t and r are set to 0, and q is set to 1. Using $glTexCoord2^*()$ allows you to specify s and t; r and q are set to 0 and 1, respectively. With $glTexCoord3^*()$, q is set to 1 and the other coordinates are set as specified. You can specify all coordinates with $glTexCoord4^*()$. Use the appropriate suffix (s, i, f, or d) and the corresponding value for TYPE (GLshort, GLint, GLfloat, or GLdouble) to specify the coordinates' data type. You can supply the coordinates individually, or you can use the vector version of the command to supply them in a single array. Texture coordinates are multiplied by the 4×4 texture matrix before any texture mapping occurs. (See "The Texture Matrix Stack.") Note that integer texture coordinates are interpreted directly rather than being mapped to the range [-1,1] as normal coordinates are.

The next section discusses how to calculate appropriate texture coordinates. Instead of explicitly assigning them yourself, you can choose to have texture coordinates calculated automatically by OpenGL as a function of the vertex coordinates. (See "Automatic Texture-Coordinate Generation.")

Computing Appropriate Texture Coordinates

Two-dimensional textures are square or rectangular images that are typically mapped to the polygons that make up a polygonal model. In the simplest case, you're mapping a rectangular texture onto a model that's also rectangular - for example, your texture is a scanned image of a brick wall, and your rectangle is to represent a brick wall of a building. Suppose the brick wall is square and the texture is square, and you want to map the whole texture to the whole wall. The texture coordinates of the texture square are (0, 0), (1, 0), (1, 1), and (0, 1) in counterclockwise order. When you're drawing the wall, just give those four coordinate sets as the texture coordinates as you specify the wall's vertices in counterclockwise order.

Now suppose that the wall is two-thirds as high as it is wide, and that the texture is again square. To avoid distorting the texture, you need to map the wall to a portion of the texture map so that the aspect ratio of the texture is preserved. Suppose that you decide to use the lower two-thirds of the texture map to texture the wall. In this case, use texture coordinates of (0,0), (1,0), (1,2/3), and (0,2/3) for the texture coordinates as the wall vertices are traversed in a counterclockwise order.

As a slightly more complicated example, suppose you'd like to display a tin can with a label wrapped around it on the screen. To obtain the texture, you purchase a can, remove the label, and scan it in. Suppose the label is 4 units tall and 12 units around, which yields an aspect ratio of 3 to 1. Since textures must have aspect ratios of 2n to 1, you can either simply not use the top third of the texture, or you can cut and paste the texture until it has the necessary aspect ratio. Suppose you decide not to use the top third. Now suppose the tin can is a cylinder approximated by thirty polygons of length 4 units (the height of the can) and width 12/30 (1/30 of the circumference of the can). You can use the following texture coordinates for each of the thirty approximating rectangles:

1: (0, 0), (1/30, 0), (1/30, 2/3), (0, 2/3)

2: (1/30, 0), (2/30, 0), (2/30, 2/3), (1/30, 2/3)

3: (2/30, 0), (3/30, 0), (3/30, 2/3), (2/30, 2/3)

• • •

30: (29/30, 0), (1, 0), (1, 2/3), (29/30, 2/3)

Only a few curved surfaces such as cones and cylinders can be mapped to a flat surface without geodesic distortion. Any other shape requires some distortion. In general, the higher the curvature of the surface, the more distortion of the texture is required.

If you don't care about texture distortion, it's often quite easy to find a reasonable mapping. For example, consider a sphere whose surface coordinates are given by (cos &thgr; cos &phgr; , cos &thgr; sin &phgr; , sin &thgr;), where 0 ≤ &thgr; ≤ 2 &pgr; , and 0 ≤ &phgr; ≤ &pgr; . The &thgr; - &phgr; rectangle can be mapped directly to a rectangular texture map, but the closer you get to the poles, the more distorted the texture is. The entire top edge of the texture map is mapped to the north pole, and the entire bottom edge to the south pole. For other surfaces, such as that of a torus (doughnut) with a large hole, the natural surface coordinates map to the texture

coordinates in a way that produces only a little distortion, so it might be suitable for many applications. Figure 9-6 shows two tori, one with a small hole (and therefore a lot of distortion near the center) and one with a large hole (and only a little distortion).

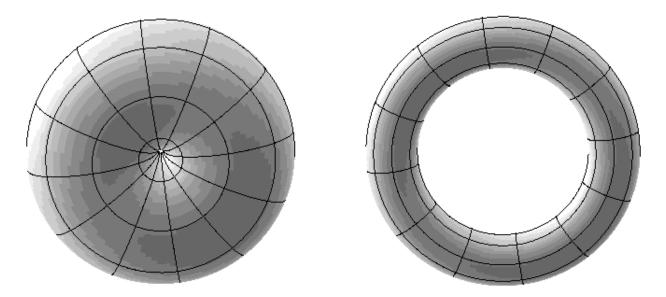


Figure 9-6 : Texture-Map Distortion

If you're texturing spline surfaces generated with evaluators (see Chapter 12), the u and v parameters for the surface can sometimes be used as texture coordinates. In general, however, there's a large artistic component to successfully mapping textures to polygonal approximations of curved surfaces.

Repeating and Clamping Textures

You can assign texture coordinates outside the range [0,1] and have them either clamp or repeat in the texture map. With repeating textures, if you have a large plane with texture coordinates running from 0.0 to 10.0 in both directions, for example, you'll get 100 copies of the texture tiled together on the screen. During repeating, the integer part of texture coordinates is ignored, and copies of the texture map tile the surface. For most applications where the texture is to be repeated, the texels at the top of the texture should match those at the bottom, and similarly for the left and right edges.

The other possibility is to clamp the texture coordinates: Any values greater than 1.0 are set to 1.0, and any values less than 0.0 are set to 0.0. Clamping is useful for applications where you want a single copy of the texture to appear on a large surface. If the surface-texture coordinates range from 0.0 to 10.0 in both directions, one copy of the texture appears in the lower corner of the surface. If you've chosen GL_LINEAR as the filtering method (see "Filtering"), an equally weighted combination of the border color and the texture color is used, as follows.

- When repeating, the 2 × 2 array wraps to the opposite edge of the texture. Thus, texels on the right edge are averaged with those on the left, and top and bottom texels are also averaged.
- If there is a border, then the texel from the border is used in the weighting. Otherwise, GL_TEXTURE_BORDER_COLOR is used. (If you've chosen GL_NEAREST as the filtering method, the border color is completely ignored.)

Note that if you are using clamping, you can avoid having the rest of the surface affected by the texture. To do this, use alpha values of 0 for the edges (or borders, if they are specified) of the texture. The decal texture function directly uses the texture's alpha value in its calculations. If you are using one of the other texture functions, you may also need to enable blending with good source and destination factors. (See "Blending" in Chapter 6.)

To see the effects of wrapping, you must have texture coordinates that venture beyond [0.0, 1.0]. Start with Example 9-1, and modify the texture coordinates for the squares by mapping the texture coordinates from 0.0 to 3.0 as follows:

```
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);
glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
glTexCoord2f(0.0, 3.0); glVertex3f(1.0, 1.0, 0.0);
glTexCoord2f(3.0, 3.0); glVertex3f(2.41421, 1.0, -1.41421);
glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421);
glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0, -1.41421); glEnd();
```

With GL_REPEAT wrapping, the result is as shown in Figure 9-7.

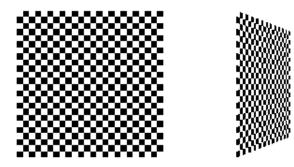


Figure 9-7 : Repeating a Texture

In this case, the texture is repeated in both the *s* and *t* directions, since the following calls are made to **glTexParameter***():

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT); glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

If GL_CLAMP is used instead of GL_REPEAT for each direction, you see something similar to Figure 9-8.



Figure 9-8 : Clamping a Texture

You can also clamp in one direction and repeat in the other, as shown in Figure 9-9.

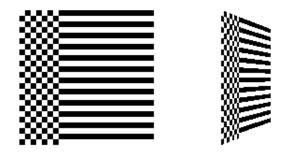


Figure 9-9 : Repeating and Clamping a Texture

You've now seen all the possible arguments for glTexParameter*(), which is summarized here.

```
void glTexParameter{if}(GLenum target, GLenum pname, TYPE param);
void glTexParameter{if}v(GLenum target, GLenum pname,
TYPE *param);
```

Sets various parameters that control how a texture is treated as it's applied to a fragment or stored in a texture object. The target parameter is either GL_TEXTURE_2D or GL_TEXTURE_1D to indicate a two- or one-dimensional texture. The possible values for pname and param are shown in Table 9-4. You can use the vector version of the command to supply an array of values for GL_TEXTURE_BORDER_COLOR, or you can supply individual values for other parameters using the nonvector version. If these values are supplied as integers, they're converted to floating-point according to Table 4-1; they're also clamped to the range [0,1].

 Table 9-4 : glTexParameter*() Parameters

Parameter	Values
GL_TEXTURE_WRAP_S	GL_CLAMP, GL_REPEAT
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_REPEAT
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR
GL_TEXTURE_BORDER_COLOR	any four values in [0.0, 1.0]
GL_TEXTURE_PRIORITY	[0.0, 1.0] for the current texture object

Try This

Figure 9-8 and Figure 9-9 are drawn using GL_NEAREST for the minification and magnification filter. What happens if you change the filter values to GL_LINEAR? Why?

Automatic Texture-Coordinate Generation

You can use texture mapping to make contours on your models or to simulate the reflections from an arbitrary environment on a shiny model. To achieve these effects, let OpenGL automatically generate the texture coordinates for you, rather than explicitly assigning them with **glTexCoord***(). To generate texture coordinates automatically, use the command **glTexGen**().

void glTexGen{ifd}(GLenum coord, GLenum pname, TYPEparam);

void **glTexGen**{*ifd*}**v**(*GLenum coord*, *GLenum pname*, *TYPE* **param*);

Specifies the functions for automatically generating texture coordinates. The first parameter, coord, must be GL_S, GL_T, GL_R, or GL_Q to indicate whether texture coordinate s, t, r, or q is to be generated. The pname parameter is GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, or GL_EYE_PLANE. If it's GL_TEXTURE_GEN_MODE, param is an integer (or, in the vector version of the command, points to an integer) that's either GL_OBJECT_LINEAR, GL_EYE_LINEAR, or GL_SPHERE_MAP. These symbolic constants determine which function is used to generate the texture coordinate. With either of the other possible values for pname, param is a pointer to an array of values (for the vector version) specifying parameters for the texture-generation function.

The different methods of texture-coordinate generation have different uses. Specifying the reference plane in object coordinates is best for when a texture image remains fixed to a moving object. Thus, GL_OBJECT_LINEAR would be used for putting a wood grain on a table top. Specifying the reference plane in eye coordinates (GL_EYE_LINEAR) is best for producing

dynamic contour lines on moving objects. GL_EYE_LINEAR may be used by specialists in geosciences, who are drilling for oil or gas. As the drill goes deeper into the ground, the drill may be rendered with different colors to represent the layers of rock at increasing depths. GL_SPHERE_MAP is predominantly used for environment mapping. (See "Environment Mapping.")

Creating Contours

When GL_TEXTURE_GEN_MODE and GL_OBJECT_LINEAR are specified, the generation function is a linear combination of the object coordinates of the vertex (*x*0,*y*0,*z*0,*w*0):

generated coordinate = p1x0 + p2y0 + p3z0 + p4w0

The p1, ..., p4 values are supplied as the *param* argument to **glTexGen*v()**, with *pname* set to GL_OBJECT_PLANE. With p1, ..., p4 correctly normalized, this function gives the distance from the vertex to a plane. For example, if p2 = p3 = p4 = 0 and p1 = 1, the function gives the distance between the vertex and the plane x = 0. The distance is positive on one side of the plane, negative on the other, and zero if the vertex lies on the plane.

Initially in Example 9-6, equally spaced contour lines are drawn on a teapot; the lines indicate the distance from the plane x = 0. The coefficients for the plane x = 0 are in this array:

```
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
```

Since only one property is being shown (the distance from the plane), a one-dimensional texture map suffices. The texture map is a constant green color, except that at equally spaced intervals it includes a red mark. Since the teapot is sitting on the *x*-*y* plane, the contours are all perpendicular to its base. "Plate 18" in Appendix I shows the picture drawn by the program.

In the same example, pressing the 's' key changes the parameters of the reference plane to

```
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
```

the contour stripes are parallel to the plane x + y + z = 0, slicing across the teapot at an angle, as shown in "Plate 18" in Appendix I. To restore the reference plane to its initial value, x = 0, press the 'x' key.

Example 9-6 : Automatic Texture-Coordinate Generation: texgen.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <Stdlib.h>
#include <stdlib.h>
#include <stdio.h>
#define stripeImageWidth 32
GLubyte stripeImage[4*stripeImageWidth];
static GLuint texName;
void makeStripeImage(void)
{
    int j;
    for (j = 0; j < stripeImageWidth; j++) {
}
</pre>
```

```
stripeImage[4*j] = (GLubyte) ((j<=4) ? 255 : 0);</pre>
      stripeImage[4*j+1] = (GLubyte) ((j>4) ? 255 : 0);
      stripeImage[4*j+2] = (GLubyte) 0;
      stripeImage[4*j+3] = (GLubyte) 255;
   }
}
/* planes for texture coordinate generation */
static GLfloat xequalzero[] = {1.0, 0.0, 0.0, 0.0};
static GLfloat slanted[] = {1.0, 1.0, 1.0, 0.0};
static GLfloat *currentCoeff;
static GLenum currentPlane;
static GLint currentGenMode;
void init(void)
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_SMOOTH);
   makeStripeImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
   glGenTextures(1, &texName);
  glBindTexture(GL_TEXTURE_1D, texName);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                   GL LINEAR);
   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                   GL LINEAR);
   glTexImagelD(GL TEXTURE 1D, 0, GL RGBA, stripeImageWidth, 0,
                GL RGBA, GL UNSIGNED BYTE, stripeImage);
   glTexEnvf(GL TEXTURE ENV, GL TEXTURE ENV MODE, GL MODULATE);
   currentCoeff = xequalzero;
   currentGenMode = GL OBJECT LINEAR;
   currentPlane = GL OBJECT PLANE;
   glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
   glTexGenfv(GL_S, currentPlane, currentCoeff);
   glEnable(GL_TEXTURE_GEN_S);
   glEnable(GL_TEXTURE_1D);
   glEnable(GL_CULL_FACE);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL NORMALIZE);
  glFrontFace(GL CW);
  glCullFace(GL_BACK);
   glMaterialf (GL_FRONT, GL_SHININESS, 64.0);
}
void display(void)
ł
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix ();
  glRotatef(45.0, 0.0, 0.0, 1.0);
  glBindTexture(GL_TEXTURE_1D, texName);
  glutSolidTeapot(2.0);
  glPopMatrix ();
   glFlush();
}
void reshape(int w, int h)
```

```
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
   if (w \le h)
      glOrtho (-3.5, 3.5, -3.5*(GLfloat)h/(GLfloat)w,
               3.5*(GLfloat)h/(GLfloat)w, -3.5, 3.5);
   else
      glOrtho (-3.5*(GLfloat)w/(GLfloat)h,
               3.5*(GLfloat)w/(GLfloat)h, -3.5, 3.5, -3.5, 3.5);
   glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
void keyboard (unsigned char key, int x, int y)
   switch (key) {
     case 'e':
      case `E':
         currentGenMode = GL_EYE_LINEAR;
         currentPlane = GL_EYE_PLANE;
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 'o':
      case '0':
         currentGenMode = GL OBJECT LINEAR;
         currentPlane = GL OBJECT PLANE;
         glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, currentGenMode);
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case `s':
      case `S':
         currentCoeff = slanted;
         glTexGenfv(GL S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case `x':
      case `X':
         currentCoeff = xequalzero;
         glTexGenfv(GL_S, currentPlane, currentCoeff);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(256, 256);
  glutInitWindowPosition(100, 100);
  glutCreateWindow (argv[0]);
   init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
   glutMainLoop();
```

```
return 0;
}
```

You enable texture-coordinate generation for the *s* coordinate by passing GL_TEXTURE_GEN_S to **glEnable**(). To generate other coordinates, enable them with GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R, or GL_TEXTURE_GEN_Q. Use **glDisable**() with the appropriate constant to disable coordinate generation. Also note the use of GL_REPEAT to cause the contour lines to be repeated across the teapot.

The GL_OBJECT_LINEAR function calculates the texture coordinates in the model's coordinate system. Initially in Example 9-6, the GL_OBJECT_LINEAR function is used, so the contour lines remain perpendicular to the base of the teapot, no matter how the teapot is rotated or viewed. However, if you press the 'e' key, the texture generation mode is changed from GL_OBJECT_LINEAR to GL_EYE_LINEAR, and the contour lines are calculated relative to the eye coordinate system. (Pressing the 'o' key restores GL_OBJECT_LINEAR as the texture generation mode.) If the reference plane is x = 0, the result is a teapot with red stripes parallel to the *y*-*z* plane from the eye's point of view, as shown in "Plate 18" in Appendix I. Mathematically, you are multiplying the vector (p1p2p3p4) by the inverse of the modelview matrix to obtain the values used to calculate the distance to the plane. The texture coordinate is generated with the following function:

generated coordinate = p1'xe + p2'ye + p3'ze + p4'we

where (p1' p2' p3' p4') = (p1p2p3p4)**M**-1

In this case, (*xe*, *ye*, *ze*, *we*) are the eye coordinates of the vertex, and p1, ..., p4 are supplied as the *param* argument to **glTexGen*(**) with *pname* set to GL_EYE_PLANE. The primed values are calculated only at the time they're specified so this operation isn't as computationally expensive as it looks.

In all these examples, a single texture coordinate is used to generate contours. The *s* and *t* texture coordinates can be generated independently, however, to indicate the distances to two different planes. With a properly constructed two-dimensional texture map, the resulting two sets of contours can be viewed simultaneously. For an added level of complexity, you can calculate the *s* coordinate using GL_OBJECT_LINEAR and the *t* coordinate using GL_EYE_LINEAR.

Environment Mapping

The goal of environment mapping is to render an object as if it were perfectly reflective, so that the colors on its surface are those reflected to the eye from its surroundings. In other words, if you look at a perfectly polished, perfectly reflective silver object in a room, you see the walls, floor, and other objects in the room reflected off the object. (A classic example of using environment mapping is the evil, morphing cyborg in the film *Terminator 2*.) The objects whose reflections you see depend on the position of your eye and on the position and surface angles of the silver object. To perform environment mapping, all you have to do is create an appropriate texture map and then have OpenGL generate the texture coordinates for you.

Environment mapping is an approximation based on the assumption that the items in the environment are far away compared to the surfaces of the shiny object - that is, it's a small object in a large room. With this assumption, to find the color of a point on the surface, take the ray from the eye to the surface, and reflect the ray off the surface. The direction of the reflected ray completely determines the color to be painted there. Encoding a color for each direction on a flat texture map is

equivalent to putting a polished perfect sphere in the middle of the environment and taking a picture of it with a camera that has a lens with a very long focal length placed far away. Mathematically, the lens has an infinite focal length and the camera is infinitely far away. The encoding therefore covers a circular region of the texture map, tangent to the top, bottom, left, and right edges of the map. The texture values outside the circle make no difference, as they are never accessed in environment mapping.

To make a perfectly correct environment texture map, you need to obtain a large silvered sphere, take a photograph of it in some environment with a camera located an infinite distance away and with a lens that has an infinite focal length, and scan in the photograph. To approximate this result, you can use a scanned-in photograph of an environment taken with an extremely wide-angle (or fish-eye) lens. Plate 21 shows a photograph taken with such a lens and the results when that image is used as an environment map.

Once you've created a texture designed for environment mapping, you need to invoke OpenGL's environment-mapping algorithm. This algorithm finds the point on the surface of the sphere with the same tangent surface as the point on the object being rendered, and it paints the object's point with the color visible on the sphere at the corresponding point.

To automatically generate the texture coordinates to support environment mapping, use this code in your program:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

The GL_SPHERE_MAP constant creates the proper texture coordinates for the environment mapping. As shown, you need to specify it for both the *s* and *t* directions. However, you don't have to specify any parameters for the texture-coordinate generation function.

The GL_SPHERE_MAP texture function generates texture coordinates using the following mathematical steps.

- 1. **u** is the unit vector pointing from the origin to the vertex (in eye coordinates).
- 2. **n'** is the current normal vector, after transformation to eye coordinates.
- 3. **r** is the reflection vector, (rxryrz)T, which is calculated by **u** 2**n'n'Tu**.
- 4. Then an interim value, *m*, is calculated by

$$m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$$

1. Finally, the *s* and *t* texture coordinates are calculated by

 $s = r_x / m + \frac{1}{2}$

Advanced Features

Advanced

This section describes how to manipulate the texture matrix stack and how to use the q coordinate. Both techniques are considered advanced, since you don't need them for many applications of texture mapping.

The Texture Matrix Stack

Just as your model coordinates are transformed by a matrix before being rendered, texture coordinates are multiplied by a 4×4 matrix before any texture mapping occurs. By default, the texture matrix is the identity, so the texture coordinates you explicitly assign or those that are automatically generated remain unchanged. By modifying the texture matrix while redrawing an object, however, you can make the texture slide over the surface, rotate around it, stretch and shrink, or any combination of the three. In fact, since the texture matrix is a completely general 4×4 matrix, effects such as perspective can be achieved.

When the four texture coordinates (s, t, r, q) are multiplied by the texture matrix, the resulting vector (s' t' r' q') is interpreted as homogeneous texture coordinates. In other words, the texture map is indexed by s'/q' and t'/q'. (Remember that r'/q' is ignored in standard OpenGL, but may be used by implementations that support a 3D texture extension.) The texture matrix is actually the top matrix on a stack, which must have a stack depth of at least two matrices. All the standard matrix-manipulation commands such as **glPushMatrix**(), **glPopMatrix**(), **glMultMatrix**(), and **glRotate***() can be applied to the texture matrix. To modify the current texture matrix, you need to set the matrix mode to GL_TEXTURE, as follows:

```
glMatrixMode(GL_TEXTURE); /* enter texture matrix mode */
glRotated(...);
/* ... other matrix manipulations ... */
glMatrixMode(GL_MODELVIEW); /* back to modelview mode */
```

The q Coordinate

The mathematics of the q coordinate in a general four-dimensional texture coordinate is as described in the previous section. You can make use of q in cases where more than one projection or perspective transformation is needed. For example, suppose you want to model a spotlight that has some nonuniform pattern - brighter in the center, perhaps, or noncircular, because of flaps or lenses that modify the shape of the beam. You can emulate shining such a light on a flat surface by making a texture map that corresponds to the shape and intensity of a light, and then projecting it on the surface in question using projection transformations. Projecting the cone of light onto surfaces in the scene requires a perspective transformation (q ≠ 1), since the lights might shine on surfaces that aren't perpendicular to them. A second perspective transformation occurs because the

viewer sees the scene from a different (but perspective) point of view. (See "Plate 27" in Appendix I for an example, and see "Fast Shadows and Lighting Effects Using Texture Mapping" by Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli, SIGGRAPH 1992 Proceedings, (*Computer Graphics*, 26:2, July 1992, p. 249-252) for more details.)

Another example might arise if the texture map to be applied comes from a photograph that itself was taken in perspective. As with spotlights, the final view depends on the combination of two perspective transformations.

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 10 The Framebuffer

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand what buffers make up the framebuffer and how they're used
- Clear selected buffers and enable them for writing
- Control the parameters of the scissoring, alpha, stencil, and depth-buffer tests that are applied to pixels
- Perform dithering and logical operations
- Use the accumulation buffer for such purposes as scene antialiasing

An important goal of almost every graphics program is to draw pictures on the screen. The screen is composed of a rectangular array of pixels, each capable of displaying a tiny square of color at that point in the image. After the rasterization stage (including texturing and fog), the data are not yet pixels, but are fragments. Each fragment has coordinate data which corresponds to a pixel, as well as color and depth values. Then each fragment undergoes a series of tests and operations, some of which have been previously described (See "Blending" in Chapter 6) and others that are discussed in this chapter.

If the tests and operations are survived, the fragment values are ready to become pixels. To draw these pixels, you need to know what color they are, which is the information that's stored in the color buffer. Whenever data is stored uniformly for each pixel, such storage for all the pixels is called a *buffer*. Different buffers might contain different amounts of data per pixel, but within a given buffer, each pixel is assigned the same amount of data. A buffer that stores a single bit of information about pixels is called a bitplane.

As shown in Figure 10-1, the lower-left pixel in an OpenGL window is pixel (0, 0), corresponding to the window coordinates of the lower-left corner of the 1×1 region occupied by this pixel. In general, pixel (*x*, *y*) fills the region bounded by *x* on the left, *x*+1 on the right, *y* on the bottom, and *y*+1 on the top.

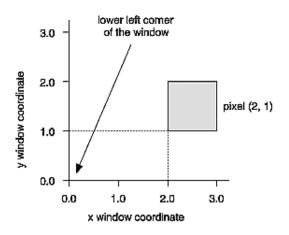


Figure 10-1 : Region Occupied by a Pixel

As an example of a buffer, let's look more closely at the color buffer, which holds the color information that's to be displayed on the screen. Assume that the screen is 1280 pixels wide and 1024 pixels high and that it's a full 24-bit color screen - in other words, there are 224 (or 16,777,216) different colors that can be displayed. Since 24 bits translates to 3 bytes (8 bits/byte), the color buffer in this example has to store at least 3 bytes of data for each of the 1,310,720 (1280*1024) pixels on the screen. A particular hardware system might have more or fewer pixels on the physical screen as well as more or less color data per pixel. Any particular color buffer, however, has the same amount of data saved for each pixel on the screen.

The color buffer is only one of several buffers that hold information about a pixel. For example, in "A Hidden-Surface Removal Survival Kit" in Chapter 5, you learned that the depth buffer holds depth information for each pixel. The color buffer itself can consist of several subbuffers. The framebuffer on a system comprises all of these buffers. With the exception of the color buffer(s), you don't view these other buffers directly; instead, you use them to perform such tasks as hidden-surface elimination, antialiasing of an entire scene, stenciling, drawing smooth motion, and other operations.

This chapter describes all the buffers that can exist in an OpenGL implementation and how they're used. It also discusses the series of tests and pixel operations that are performed before any data is written to the viewable color buffer. Finally, it explains how to use the accumulation buffer, which is used to accumulate images that are drawn into the color buffer. This chapter has the following major sections.

- "Buffers and Their Uses" describes the possible buffers, what they're for, and how to clear them and enable them for writing.
- "Testing and Operating on Fragments" explains the scissoring, alpha, stencil, and depth-buffer tests that occur after a pixel's position and color have been calculated but before this information is drawn on the screen. Several operations blending, dithering, and logical operations can also be performed before a fragment updates the screen.
- "The Accumulation Buffer" describes how to perform several advanced techniques using the accumulation buffer. These techniques include antialiasing an entire scene, using motion blur, and simulating photographic depth of field.

Buffers and Their Uses

An OpenGL system can manipulate the following buffers:

- Color buffers: front-left, front-right, back-left, back-right, and any number of auxiliary color buffers
- Depth buffer
- Stencil buffer
- Accumulation buffer

Your particular OpenGL implementation determines which buffers are available and how many bits per pixel each holds. Additionally, you can have multiple visuals, or window types, that have different buffers available. Table 10-1 lists the parameters to use with **glGetIntegerv()** to query your OpenGL system about per-pixel buffer storage for a particular visual.

Note: If you're using the X Window System, you're guaranteed, at a minimum, to have a visual with one color buffer for use in RGBA mode with associated stencil, depth, and accumulation buffers that have color components of nonzero size. Also, if your X Window System implementation supports a Pseudo-Color visual, you are also guaranteed to have one OpenGL visual that has a color buffer for use in color-index mode with associated depth and stencil buffers. You'll probably want to use **glXGetConfig**() to query your visuals; see Appendix C and the *OpenGL Reference Manual* for more information about this routine.

Parameter	Meaning
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	Number of bits per R, G, B, or A component in the color buffers
GL_INDEX_BITS	Number of bits per index in the color buffers
GL_DEPTH_BITS	Number of bits per pixel in the depth buffer
GL_STENCIL_BITS	Number of bits per pixel in the stencil buffer
GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, GL_ACCUM_ALPHA_BITS	Number of bits per R, G, B, or A component in the accumulation buffer

Table 10-1 : Query Parameters for Per-Pixel Buffer Storage

Color Buffers

The color buffers are the ones to which you usually draw. They contain either color-index or RGB color data and may also contain alpha values. An OpenGL implementation that supports stereoscopic viewing has left and right color buffers for the left and right stereo images. If stereo isn't supported, only the left buffers are used. Similarly, double-buffered systems have front and back buffers, and a single-buffered system has the front buffers only. Every OpenGL implementation must provide a front-left color buffer.

Optional, nondisplayable auxiliary color buffers may also be supported. OpenGL doesn't specify any particular uses for these buffers, so you can define and use them however you please. For example, you might use them for saving an image that you use repeatedly. Then rather than redrawing the image, you can just copy it from an auxiliary buffer into the usual color buffers. (See the description of **glCopyPixels**() in "Reading, Writing, and Copying Pixel Data" in Chapter 8 for more information about how to do this.)

You can use GL_STEREO or GL_DOUBLEBUFFER with **glGetBooleanv**() to find out if your system supports stereo (that is, has left and right buffers) or double-buffering (has front and back buffers). To find out how many, if any, auxiliary buffers are present, use **glGetIntegerv**() with GL_AUX_BUFFERS.

Depth Buffer

The depth buffer stores a depth value for each pixel. As described in "A Hidden-Surface Removal Survival Kit" in Chapter 5, depth is usually measured in terms of distance to the eye, so pixels with larger depth-buffer values are overwritten by pixels with smaller values. This is just a useful convention, however, and the depth buffer's behavior can be modified as described in "Depth Test." The depth buffer is sometimes called the *z buffer* (the z comes from the fact that *x* and *y* values measure horizontal and vertical displacement on the screen, and the *z* value measures distance perpendicular to the screen).

Stencil Buffer

One use for the stencil buffer is to restrict drawing to certain portions of the screen, just as a cardboard stencil can be used with a can of spray paint to make fairly precise painted images. For example, if you want to draw an image as it would appear through an odd-shaped windshield, you can store an image of the windshield's shape in the stencil buffer, and then draw the entire scene. The stencil buffer prevents anything that wouldn't be visible through the windshield from being drawn. Thus, if your application is a driving simulation, you can draw all the instruments and other items inside the automobile once, and as the car moves, only the outside scene need be updated.

Accumulation Buffer

The accumulation buffer holds RGBA color data just like the color buffers do in RGBA mode. (The results of using the accumulation buffer in color-index mode are undefined.) It's typically used for accumulating a series of images into a final, composite image. With this method, you can perform operations like scene antialiasing by supersampling an image and then averaging the samples to produce the values that are finally painted into the pixels of the color buffers. You don't draw directly into the accumulation buffer; accumulation operations are always performed in rectangular blocks, which are usually transfers of data to or from a color buffer.

Clearing Buffers

In graphics programs, clearing the screen (or any of the buffers) is typically one of the most expensive operations you can perform - on a 1280×1024 monitor, it requires touching well over a million pixels. For simple graphics applications, the clear operation can take more time than the rest of the drawing. If you need to clear not only the color buffer but also the depth and stencil buffers, the clear operation can be three times as expensive.

To address this problem, some machines have hardware that can clear more than one buffer at once. The OpenGL clearing commands are structured to take advantage of architectures like this. First, you specify the values to be written into each buffer to be cleared. Then you issue a single command to perform the clear operation, passing in a list of all the buffers to be cleared. If the hardware is capable of simultaneous clears, they all occur at once; otherwise, each buffer is cleared sequentially.

The following commands set the clearing values for each buffer.

void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha); void glClearIndex(GLfloat index); void glClearDepth(GLclampd depth); void glClearStencil(GLint s); void glClearAccum(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha); Specifies the current clearing values for the color buffer (in RGBA mode), the color buffer (in color-index mode), the depth buffer, the stencil buffer, and the accumulation buffer. The GLclampf and GLclampd types (clamped GLfloat and clamped GLdouble) are clamped to be

between 0.0 and 1.0. The default depth-clearing value is 1.0; all the other default clearing values are 0. The values set with the clear commands remain in effect until they're changed by another call to the same command.

After you've selected your clearing values and you're ready to clear the buffers, use glClear().

void glClear(GLbitfield mask);

Clears the specified buffers. The value of mask is the bitwise logical OR of some combination of GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT, and GL_ACCUM_BUFFER_BIT to identify which buffers are to be cleared. GL_COLOR_BUFFER_BIT clears either the RGBA color or the color-index buffer, depending on the mode of the system at the time. When you clear the color or color-index buffer, all the color buffers that are enabled for writing (see the next section) are cleared. The pixel ownership test, scissor test, and dithering, if enabled, are applied to the clearing operation. Masking operations, such as **glColorMask(**) and **glIndexMask(**), are also effective. The alpha test, stencil test, and depth test do not affect the operation of **glClear(**).

Selecting Color Buffers for Writing and Reading

The results of a drawing or reading operation can go into or come from any of the color buffers: front, back, front-left, back-left, front-right, back-right, or any of the auxiliary buffers. You can choose an individual buffer to be the drawing or reading target. For drawing, you can also set the target to draw into more than one buffer at the same time. You use **glDrawBuffer**() to select the buffers to be written and **glReadBuffer**() to select the buffer as the source for **glReadPixels**(),

glCopyPixels(), glCopyTexImage*(), and glCopyTexSubImage*().

If you are using double-buffering, you usually want to draw only in the back buffer (and swap the buffers when you're finished drawing). In some situations, you might want to treat a double-buffered window as though it were single-buffered by calling **glDrawBuffer**() to enable you to draw to both front and back buffers at the same time.

glDrawBuffer() is also used to select buffers to render stereo images (GL*LEFT and GL*RIGHT) and to render into auxiliary buffers (GL_AUX*i*).

void glDrawBuffer(GLenum mode);

Selects the color buffers enabled for writing or clearing. Disables buffers enabled by previous calls to **glDrawBuffer()**. More than one buffer may be enabled at one time. The value of mode can be one of the following:

GL_FRONT	GL_FRONT_LEFT	GL_AUXi
GL_BACK	GL_FRONT_RIGHT	GL_FRONT_AND_BACK
GL_LEFT	GL_BACK_LEFT	GL_NONE
GL_RIGHT	GL_BACK_RIGHT	

Arguments that omit LEFT or RIGHT refer to both the left and right buffers; similarly, arguments that omit FRONT or BACK refer to both. The i in GL_AUXi is a digit identifying a particular auxiliary buffer. By default, mode is GL_FRONT for single-buffered contexts and GL_BACK for double-buffered contexts.

Note: You can enable drawing to nonexistent buffers as long as you enable drawing to at least one buffer that does exist. If none of the specified buffers exist, an error results.

void glReadBuffer(GLenum mode);

Selects the color buffer enabled as the source for reading pixels for subsequent calls to **glReadPixels()**, **glCopyPixels()**, **glCopyTexImage*()**, and **glCopyTexSubImage*()**. Disables buffers enabled by previous calls to **glReadBuffer()**. The value of mode can be one of the following:

- GL_FRONT GL_FRONT_LEFT GL_AUXi
- GL_BACK GL_FRONT_RIGHT

GL_LEFT GL_BACK_LEFT

By default, mode is GL_FRONT for single-buffered contexts and GL_BACK for double-buffered contexts.

Note: You must enable reading from a buffer that does exist or an error results.

Masking Buffers

Before OpenGL writes data into the enabled color, depth, or stencil buffers, a masking operation is applied to the data, as specified with one of the following commands. A bitwise logical AND is performed with each mask and the corresponding data to be written.

void glIndexMask(GLuint mask); void glColorMask(GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha); void glDepthMask(GLboolean flag); void glStencilMask(GLuint mask); Sets the masks used to control writing into the indicated buffers. The mask set by glIndexMask() applies only in color-index mode. If a 1 appears in mask, the corresponding bit in the color-index buffer is written: where a 0 appears the bit isn't written. Similarly

bit in the color-index buffer is written; where a 0 appears, the bit isn't written. Similarly, glColorMask() affects drawing in RGBA mode only. The red, green, blue, and alpha values control whether the corresponding component is written. (GL_TRUE means it is written.) If flag is GL_TRUE for glDepthMask(), the depth buffer is enabled for writing; otherwise, it's disabled. The mask for glStencilMask() is used for stencil data in the same way as the mask is used for color-index data in glIndexMask(). The default values of all the GLboolean masks are GL_TRUE, and the default values for the two GLuint masks are all 1's.

You can do plenty of tricks with color masking in color-index mode. For example, you can use each bit in the index as a different layer and set up interactions between arbitrary layers with appropriate settings of the color map. You can create overlays and underlays, and do so-called color-map animations. (See Chapter 14 for examples of using color masking.) Masking in RGBA mode is useful less often, but you can use it for loading separate image files into the red, green, and blue bitplanes, for example.

You've seen one use for disabling the depth buffer in "Three-Dimensional Blending with the Depth Buffer" in Chapter 6. Disabling the depth buffer for writing can also be useful if a common background is desired for a series of frames, and you want to add some features that may be obscured by parts of the background. For example, suppose your background is a forest, and you would like to draw repeated frames with the same trees, but with objects moving among them. After the trees are drawn with their depths recorded in the depth buffer, then the image of the trees is saved, and the new items are drawn with the depth buffer disabled for writing. As long as the new items don't overlap each other, the picture is correct. To draw the next frame, restore the image of the trees and continue. You don't need to restore the values in the depth buffer. This trick is most useful if the background is extremely complex - so complex that it's much faster just to recopy the image into the color buffer than to recompute it from the geometry.

Masking the stencil buffer can allow you to use a multiple-bit stencil buffer to hold multiple stencils (one per bit). You might use this technique to perform capping as explained in "Stencil

Test" or to implement the Game of Life as described in "Life in the Stencil Buffer" in Chapter 14.

Note: The mask specified by **glStencilMask**() controls which stencil bitplanes are written. This mask isn't related to the mask that's specified as the third parameter of **glStencilFunc**(), which specifies which bitplanes are considered by the stencil function.

Testing and Operating on Fragments

When you draw geometry, text, or images on the screen, OpenGL performs several calculations to rotate, translate, scale, determine the lighting, project the object(s) into perspective, figure out which pixels in the window are affected, and determine what colors those pixels should be drawn. Many of the earlier chapters in this book give some information about how to control these operations. After OpenGL determines that an individual fragment should be generated and what its color should be, several processing stages remain that control how and whether the fragment is drawn as a pixel into the framebuffer. For example, if it's outside a rectangular region or if it's farther from the viewpoint than the pixel that's already in the framebuffer, it isn't drawn. In another stage, the fragment's color is blended with the color of the pixel already in the framebuffer.

This section describes both the complete set of tests that a fragment must pass before it goes into the framebuffer and the possible final operations that can be performed on the fragment as it's written. The tests and operations occur in the following order; if a fragment is eliminated in an early test, none of the later tests or operations take place.

- 1. Scissor test
- 2. Alpha test
- 3. Stencil test
- 4. Depth test
- 5. Blending
- 6. Dithering
- 7. Logical operation

Each of these tests and operations is described in detail in the following sections.

Scissor Test

You can define a rectangular portion of your window and restrict drawing to take place within it by using the **glScissor**() command. If a fragment lies inside the rectangle, it passes the scissor test.

void glScissor(GLint x, GLint y, GLsizei width, GLsizei height);

Sets the location and size of the scissor rectangle (also known as the scissor box). The parameters define the lower-left corner (x, y), and the width and height of the rectangle. Pixels that lie inside the rectangle pass the scissor test. Scissoring is enabled and disabled by passing GL_SCISSOR_TEST to **glEnable**() and **glDisable**(). By default, the rectangle

The scissor test is just a version of a stencil test using a rectangular region of the screen. It's fairly easy to create a blindingly fast hardware implementation of scissoring, while a given system might be much slower at stenciling - perhaps because the stenciling is performed in software.

Advanced

An advanced use of scissoring is performing nonlinear projection. First divide the window into a regular grid of subregions, specifying viewport and scissor parameters that limit rendering to one region at a time. Then project the entire scene to each region using a different projection matrix.

To determine whether scissoring is enabled and to obtain the values that define the scissor rectangle, you can use GL_SCISSOR_TEST with **glIsEnabled()** and GL_SCISSOR_BOX with **glGetIntegerv()**.

Alpha Test

In RGBA mode, the alpha test allows you to accept or reject a fragment based on its alpha value. The alpha test is enabled and disabled by passing GL_ALPHA_TEST to **glEnable()** and **glDisable()**. To determine whether the alpha test is enabled, use GL_ALPHA_TEST with **glIsEnabled()**.

If enabled, the test compares the incoming alpha value with a reference value. The fragment is accepted or rejected depending on the result of the comparison. Both the reference value and the comparison function are set with **glAlphaFunc(**). By default, the reference value is zero, the comparison function is GL_ALWAYS, and the alpha test is disabled. To obtain the alpha comparison function or reference value, use GL_ALPHA_TEST_FUNC or GL_ALPHA_TEST_REF with **glGetIntegerv(**).

void glAlphaFunc(GLenum func, GLclampf ref);

Sets the reference value and comparison function for the alpha test. The reference value ref is clamped to be between zero and one. The possible values for func and their meaning are listed in Table 10-2.

Table 10-2 : glAlphaFunc() Parameter Values (continued)

Parameter	Meaning
GL_NEVER	Never accept the fragment
GL_ALWAYS	Always accept the fragment
GL_LESS	Accept fragment if fragment alpha < reference alpha
GL_LEQUAL	Accept fragment if fragment alpha ≤ reference alpha
GL_EQUAL	Accept fragment if fragment alpha = reference alpha
GL_GEQUAL	Accept fragment if fragment alpha ≥ reference alpha
GL_GREATER	Accept fragment if fragment alpha > reference alpha
GL_NOTEQUAL	Accept fragment if fragment alpha ≠ reference alpha

One application for the alpha test is to implement a transparency algorithm. Render your entire scene twice, the first time accepting only fragments with alpha values of one, and the second time accepting fragments with alpha values that aren't equal to one. Turn the depth buffer on during both passes, but disable depth buffer writing during the second pass.

Another use might be to make decals with texture maps where you can see through certain parts of the decals. Set the alphas in the decals to 0.0 where you want to see through, set them to 1.0 otherwise, set the reference value to 0.5 (or anything between 0.0 and 1.0), and set the comparison function to GL_GREATER. The decal has see-through parts, and the values in the depth buffer aren't affected. This technique, called billboarding, is described in "Sample Uses of Blending" in Chapter 6.

Stencil Test

The stencil test takes place only if there is a stencil buffer. (If there is no stencil buffer, the stencil test always passes.) Stenciling applies a test that compares a reference value with the value stored at a pixel in the stencil buffer. Depending on the result of the test, the value in the stencil buffer is modified. You can choose the particular comparison function used, the reference value, and the modification performed with the **glStencilFunc**() and **glStencilOp**() commands.

void glStencilFunc(GLenum func, GLint ref, GLuint mask);

Sets the comparison function (func), reference value (ref), and a mask (mask) for use with the stencil test. The reference value is compared to the value in the stencil buffer using the comparison function, but the comparison applies only to those bits where the corresponding bits of the mask are 1. The function can be GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL. If it's GL_LESS, for example, then the fragment passes if ref is less than the value in the stencil buffer. If the stencil buffer contains s bitplanes, the low-order s bits of mask are bitwise ANDed with the value in the stencil buffer and with the reference value before the comparison

is performed. The masked values are all interpreted as nonnegative values. The stencil test is enabled and disabled by passing GL_STENCIL_TEST to **glEnable()** and **glDisable()**. By default, func is GL_ALWAYS, ref is 0, mask is all 1's, and stenciling is disabled.

void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
Specifies how the data in the stencil buffer is modified when a fragment passes or fails the stencil test. The three functions fail, zfail, and zpass can be GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, or GL_INVERT. They correspond to keeping the current value, replacing it with zero, replacing it with the reference value, incrementing it, decrementing it, and bitwise-inverting it. The result of the increment and decrement functions is clamped to lie between zero and the maximum unsigned integer value (2s-1 if the stencil buffer holds s bits). The fail function is applied if the fragment fails the stencil test; if it passes, then zfail is applied if the depth test fails and zpass if the depth test passes, or if no depth test is performed. (See "Depth Test.") By default, all three stencil operations are GL_KEEP.

Stencil Queries

You can obtain the values for all six stencil-related parameters by using the query function **glGetIntegerv()** and one of the values shown in Table 10-3. You can also determine whether the stencil test is enabled by passing GL_STENCIL_TEST to **glIsEnabled()**.

Query Value	Meaning
GL_STENCIL_FUNC	Stencil function
GL_STENCIL_REF	Stencil reference value
GL_STENCIL_VALUE_MASK	Stencil mask
GL_STENCIL_FAIL	Stencil fail action
GL_STENCIL_PASS_DEPTH_FAIL	Stencil pass and depth buffer fail action
GL_STENCIL_PASS_DEPTH_PASS	Stencil pass and depth buffer pass action

 Table 10-3 : Query Values for the Stencil Test (continued)

Stencil Examples

Probably the most typical use of the stencil test is to mask out an irregularly shaped region of the screen to prevent drawing from occurring within it (as in the windshield example in "Buffers and Their Uses"). To do this, fill the stencil mask with zeros, and then draw the desired shape in the stencil buffer with 1's. You can't draw geometry directly into the stencil buffer, but you can achieve the same result by drawing into the color buffer and choosing a suitable value for the *zpass* function (such as GL_REPLACE). (You can use **glDrawPixels**() to draw pixel data directly into the stencil buffer.) Whenever drawing occurs, a value is also written into the stencil buffer (in this case, the reference value). To prevent the stencil-buffer drawing from affecting the contents of the color

buffer, set the color mask to zero (or GL_FALSE). You might also want to disable writing into the depth buffer.

After you've defined the stencil area, set the reference value to one, and the comparison function such that the fragment passes if the reference value is equal to the stencil-plane value. During drawing, don't modify the contents of the stencil planes.

Example 10-1 demonstrates how to use the stencil test in this way. Two tori are drawn, with a diamond-shaped cutout in the center of the scene. Within the diamond-shaped stencil mask, a sphere is drawn. In this example, drawing into the stencil buffer takes place only when the window is redrawn, so the color buffer is cleared after the stencil mask has been created.

Example 10-1 : Using the Stencil Test: stencil.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#define YELLOWMAT
                    1
#define BLUEMAT 2
void init (void)
   GLfloat yellow_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
  GLfloat yellow_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat blue_diffuse[] = { 0.1, 0.1, 0.7, 1.0 };
   GLfloat blue_specular[] = { 0.1, 1.0, 1.0, 1.0 };
   GLfloat position_one[] = { 1.0, 1.0, 1.0, 0.0 };
   glNewList(YELLOWMAT, GL COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, yellow_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, yellow_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 64.0);
   glEndList();
   glNewList(BLUEMAT, GL COMPILE);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, blue_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, blue_specular);
   glMaterialf(GL_FRONT, GL_SHININESS, 45.0);
   glEndList();
   glLightfv(GL_LIGHT0, GL_POSITION, position_one);
   glEnable(GL LIGHT0);
   glEnable(GL LIGHTING);
   glEnable(GL_DEPTH_TEST);
   glClearStencil(0x0);
  glEnable(GL_STENCIL_TEST);
}
/* Draw a sphere in a diamond-shaped section in the
 * middle of a window with 2 tori.
 * /
void display(void)
  glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
```

```
/* draw blue sphere where the stencil is 1 */
   glStencilFunc (GL_EQUAL, 0x1, 0x1);
   glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
   glCallList (BLUEMAT);
   glutSolidSphere (0.5, 15, 15);
/* draw the tori where the stencil is not 1 */
   glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
   glPushMatrix();
      glRotatef (45.0, 0.0, 0.0, 1.0);
      glRotatef (45.0, 0.0, 1.0, 0.0);
      glCallList (YELLOWMAT);
      glutSolidTorus (0.275, 0.85, 15, 15);
      glPushMatrix();
         glRotatef (90.0, 1.0, 0.0, 0.0);
         glutSolidTorus (0.275, 0.85, 15, 15);
      glPopMatrix();
  glPopMatrix();
}
/*
   Whenever the window is reshaped, redefine the
 *
   coordinate system and redraw the stencil area.
 * /
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
/* create a diamond shaped stencil area */
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
   if (w \le h)
      gluOrtho2D(-3.0, 3.0, -3.0*(GLfloat)h/(GLfloat)w,
                 3.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h,
                 3.0*(GLfloat)w/(GLfloat)h, -3.0, 3.0);
   glMatrixMode(GL MODELVIEW);
   glLoadIdentity();
   glClear(GL_STENCIL_BUFFER_BIT);
   glStencilFunc (GL_ALWAYS, 0x1, 0x1);
   glStencilOp (GL_REPLACE, GL_REPLACE, GL_REPLACE);
   glBegin(GL_QUADS);
      glVertex2f (-1.0, 0.0);
      glVertex2f (0.0, 1.0);
      glVertex2f (1.0, 0.0);
      glVertex2f (0.0, -1.0);
   glEnd();
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0, (GLfloat) w/(GLfloat) h, 3.0, 7.0);
   glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0, 0.0, -5.0);
}
/* Main Loop
 * Be certain to request stencil bits.
 */
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                         | GLUT_DEPTH | GLUT_STENCIL);
```

```
glutInitWindowSize (400, 400);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutReshapeFunc(reshape);
glutDisplayFunc(display);
glutMainLoop();
return 0;
```

The following examples illustrate other uses of the stencil test. (See Chapter 14 for additional ideas.)

- Capping Suppose you're drawing a closed convex object (or several of them, as long as they don't intersect or enclose each other) made up of several polygons, and you have a clipping plane that may or may not slice off a piece of it. Suppose that if the plane does intersect the object, you want to cap the object with some constant-colored surface, rather than seeing the inside of it. To do this, clear the stencil buffer to zeros, and begin drawing with stenciling enabled and the stencil comparison function set to always accept fragments. Invert the value in the stencil planes each time a fragment is accepted. After all the objects are drawn, regions of the screen where no capping is required have zeros in the stencil planes, and regions requiring capping are nonzero. Reset the stencil function so that it draws only where the stencil value is nonzero, and draw a large polygon of the capping color across the entire screen.
- Overlapping translucent polygons Suppose you have a translucent surface that's made up of polygons that overlap slightly. If you simply use alpha blending, portions of the underlying objects are covered by more than one transparent surface, which doesn't look right. Use the stencil planes to make sure that each fragment is covered by at most one portion of the transparent surface. Do this by clearing the stencil planes to zeros, drawing only when the stencil plane is zero, and incrementing the value in the stencil plane when you draw.
- Stippling Suppose you want to draw an image with a stipple pattern. (See "Displaying Points, Lines, and Polygons" in Chapter 2 for more information about stippling.) You can do this by writing the stipple pattern into the stencil buffer, and then drawing conditionally on the contents of the stencil buffer. After the original stipple pattern is drawn, the stencil buffer isn't altered while drawing the image, so the object gets stippled by the pattern in the stencil planes.

Depth Test

}

For each pixel on the screen, the depth buffer keeps track of the distance between the viewpoint and the object occupying that pixel. Then if the specified depth test passes, the incoming depth value replaces the one already in the depth buffer.

The depth buffer is generally used for hidden-surface elimination. If a new candidate color for that pixel appears, it's drawn only if the corresponding object is closer than the previous object. In this way, after the entire scene has been rendered, only objects that aren't obscured by other items remain. Initially, the clearing value for the depth buffer is a value that's as far from the viewpoint as possible, so the depth of any object is nearer than that value. If this is how you want to use the depth buffer, you simply have to enable it by passing GL_DEPTH_TEST to **glEnable()** and remember to clear the depth buffer before you redraw each frame. (See "Clearing Buffers.") You can also choose a different comparison function for the depth test with **glDepthFunc()**.

void glDepthFunc(GLenum func);

Sets the comparison function for the depth test. The value for func must be GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GEQUAL, GL_GREATER, or GL_NOTEQUAL. An incoming fragment passes the depth test if its z value has the specified relation to the value already stored in the depth buffer. The default is GL_LESS, which means that an incoming fragment passes the test if its z value is less than that already stored in the depth buffer. In this case, the z value represents the distance from the object to the viewpoint, and smaller values mean the corresponding objects are closer to the viewpoint.

Blending, Dithering, and Logical Operations

Once an incoming fragment has passed all the tests described in the previous section, it can be combined with the current contents of the color buffer in one of several ways. The simplest way, which is also the default, is to overwrite the existing values. Alternatively, if you're using RGBA mode and you want the fragment to be translucent or antialiased, you might average its value with the value already in the buffer (blending). On systems with a small number of available colors, you might want to dither color values to increase the number of colors available at the cost of a loss in resolution. In the final stage, you can use arbitrary bitwise logical operations to combine the incoming fragment and the pixel that's already written.

Blending

Blending combines the incoming fragment's R, G, B, and alpha values with those of the pixel already stored at the location. Different blending operations can be applied, and the blending that occurs depends on the values of the incoming alpha value and the alpha value (if any) stored at the pixel. (See "Blending" in Chapter 6 for an extensive discussion of this topic.)

Dithering

On systems with a small number of color bitplanes, you can improve the color resolution at the expense of spatial resolution by dithering the color in the image. Dithering is like halftoning in newspapers. Although The New York Times has only two colors - black and white - it can show photographs by representing the shades of gray with combinations of black and white dots. Comparing a newspaper image of a photo (having no shades of gray) with the original photo (with grayscale) makes the loss of spatial resolution obvious. Similarly, systems with a small number of color bitplanes may dither values of red, green, and blue on neighboring pixels for the perception of a wider range of colors.

The dithering operation that takes place is hardware-dependent; all OpenGL allows you to do is to turn it on and off. In fact, on some machines, enabling dithering might do nothing at all, which makes sense if the machine already has high color resolution. To enable and disable dithering, pass GL_DITHER to **glEnable()** and **glDisable()**. Dithering is enabled by default.

Dithering applies in both RGBA and color-index mode. The colors or color indices alternate in some hardware-dependent way between the two nearest possibilities. For example, in color-index mode, if dithering is enabled and the color index to be painted is 4.4, then 60% of the pixels may be painted with index 4 and 40% of the pixels with index 5. (Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend upon only the incoming value and the fragment's x and y coordinates.) In RGBA mode, dithering is performed separately for each component (including alpha). To use dithering in color-index mode, you generally need to arrange the colors in the color map appropriately in ramps, otherwise, bizarre images might result.

Logical Operations

The final operation on a fragment is the logical operation, such as an OR, XOR, or INVERT, which is applied to the incoming fragment values (source) and/or those currently in the color buffer (destination). Such fragment operations are especially useful on bit-blt-type machines, on which the primary graphics operation is copying a rectangle of data from one place in the window to another, from the window to processor memory, or from memory to the window. Typically, the copy doesn't write the data directly into memory but instead allows you to perform an arbitrary logical operation on the incoming data and the data already present; then it replaces the existing data with the results of the operation.

Since this process can be implemented fairly cheaply in hardware, many such machines are available. As an example of using a logical operation, XOR can be used to draw on an image in an undoable way; simply XOR the same drawing again, and the original image is restored. As another example, when using color-index mode, the color indices can be interpreted as bit patterns. Then you can compose an image as combinations of drawings on different layers, use writemasks to limit drawing to different sets of bitplanes, and perform logical operations to modify different layers.

You enable and disable logical operations by passing GL_INDEX_LOGIC_OP or GL_COLOR_LOGIC_OP to **glEnable**() and **glDisable**() for color-index mode or RGBA mode, respectively. You also must choose among the sixteen logical operations with **glLogicOp**(), or you'll just get the effect of the default value, GL_COPY. (For backward compatibility with OpenGL Version 1.0, **glEnable**(GL_LOGIC_OP) also enables logical operation in color-index mode.)

void glLogicOp(GLenum opcode);

Selects the logical operation to be performed, given an incoming (source) fragment and the pixel currently stored in the color buffer (destination). Table 10-4 shows the possible values for opcode and their meaning (s represents source and d destination). The default value is GL_COPY.

Table 10-4 : Sixteen Logical Operations

Parameter	Operation	Parameter	Operation
GL_CLEAR	0	GL_AND	s ∧ d
GL_COPY	S	GL_OR	s ∨ d
GL_NOOP	d	GL_NAND	\neg (s ∧ d)
GL_SET	1	GL_NOR	\neg (s ∨ d)
GL_COPY_INVERTED	$\neg s$	GL_XOR	s XOR d
GL_INVERT	$\neg d$	GL_EQUIV	$\neg(s XOR d)$
GL_AND_REVERSE	s & and; $\neg d$	GL_AND_INVERTED	$\neg s \& and; d$
GL_OR_REVERSE	s ∨ ¬d	GL_OR_INVERTED	$\neg s \& or; d$

The Accumulation Buffer

Advanced

The accumulation buffer can be used for such things as scene antialiasing, motion blur, simulating photographic depth of field, and calculating the soft shadows that result from multiple light sources. Other techniques are possible, especially in combination with some of the other buffers. (See *The Accumulation Buffer: Hardware Support for High-Quality Rendering* by Paul Haeberli and Kurt Akeley (SIGGRAPH 1990 Proceedings, p. 309-318) for more information on the uses for the accumulation buffer.)

OpenGL graphics operations don't write directly into the accumulation buffer. Typically, a series of images is generated in one of the standard color buffers, and these are accumulated, one at a time, into the accumulation buffer. When the accumulation is finished, the result is copied back into a color buffer for viewing. To reduce rounding errors, the accumulation buffer may have higher precision (more bits per color) than the standard color buffers. Rendering a scene several times obviously takes longer than rendering it once, but the result is higher quality. You can decide what trade-off between quality and rendering time is appropriate for your application.

You can use the accumulation buffer the same way a photographer can use film for multiple exposures. A photographer typically creates a multiple exposure by taking several pictures of the same scene without advancing the film. If anything in the scene moves, that object appears blurred. Not surprisingly, a computer can do more with an image than a photographer can do with a camera. For example, a computer has exquisite control over the viewpoint, but a photographer can't shake a camera a predictable and controlled amount. (See "Clearing Buffers" for information about how to clear the accumulation buffer; use **glAccum**() to control it.)

void glAccum(GLenum op, GLfloat value);

Controls the accumulation buffer. The op parameter selects the operation, and value is a number to be used in that operation. The possible operations are GL_ACCUM, GL_LOAD, GL_RETURN, GL_ADD, and GL_MULT.

- GL_ACCUM reads each pixel from the buffer currently selected for reading with **glReadBuffer**(), multiplies the R, G, B, and alpha values by *value*, and adds the result to the accumulation buffer.
- GL_LOAD does the same thing, except that the values replace those in the accumulation buffer rather than being added to them.
- GL_RETURN takes values from the accumulation buffer, multiplies them by *value*, and places the result in the color buffer(s) enabled for writing.
- GL_ADD and GL_MULT simply add or multiply the value of each pixel in the accumulation buffer by *value* and then return it to the accumulation buffer. For GL_MULT, *value* is clamped to be in the range [-1.0,1.0]. For GL_ADD, no clamping occurs.

Scene Antialiasing

To perform scene antialiasing, first clear the accumulation buffer and enable the front buffer for reading and writing. Then loop several times (say, n) through code that jitters and draws the image (jittering is moving the image to a slightly different position), accumulating the data with

 $glAccum(GL_ACCUM, 1.0/n);$

and finally calling

```
glAccum(GL_RETURN, 1.0);
```

Note that this method is a bit faster if, on the first pass through the loop, GL_LOAD is used and clearing the accumulation buffer is omitted. See Table 10-5 for possible jittering values. With this code, the image is drawn *n* times before the final image is drawn. If you want to avoid showing the user the intermediate images, draw into a color buffer that's not displayed, accumulate from that, and use the GL_RETURN call to draw into a displayed buffer (or into a back buffer that you subsequently swap to the front).

You could instead present a user interface that shows the viewed image improving as each additional piece is accumulated and that allows the user to halt the process when the image is good enough. To accomplish this, in the loop that draws successive images, call **glAccum()** with GL_RETURN after each accumulation, using 16.0/1.0, 16.0/2.0, 16.0/3.0, ... as the second argument. With this technique, after one pass, 1/16 of the final image is shown, after two passes, 2/16 is shown, and so on. After the GL_RETURN, the code should check to see if the user wants to interrupt the process. This interface is slightly slower, since the resultant image must be copied in after each pass.

To decide what *n* should be, you need to trade off speed (the more times you draw the scene, the longer it takes to obtain the final image) and quality (the more times you draw the scene, the smoother it gets, until you make maximum use of the accumulation buffer's resolution). "Plate 22" and "Plate 23" show improvements made using scene antialiasing.

Example 10-2 defines two routines for jittering that you might find useful: **accPerspective(**) and **accFrustum(**). The routine **accPerspective(**) is used in place of **gluPerspective(**), and the first four parameters of both routines are the same. To jitter the viewing frustum for scene antialiasing, pass the *x* and *y* jitter values (of less than one pixel) to the fifth and sixth parameters of **accPerspective(**). Also pass 0.0 for the seventh and eighth parameters to **accPerspective(**) and a nonzero value for the ninth parameter (to prevent division by zero inside **accPerspective(**)). These last three parameters are used for depth-of-field effects, which are described later in this chapter.

Example 10-2 : Routines for Jittering the Viewing Volume: accpersp.c

```
#define PI 3.14159265358979323846
void accFrustum(GLdouble left, GLdouble right, GLdouble bottom,
    GLdouble top, GLdouble near, GLdouble far, GLdouble pixdx,
    GLdouble pixdy, GLdouble eyedx, GLdouble eyedy,
   GLdouble focus)
{
   GLdouble xwsize, ywsize;
   GLdouble dx, dy;
   GLint viewport[4];
   glGetIntegerv (GL_VIEWPORT, viewport);
   xwsize = right - left;
    ywsize = top - bottom;
   dx = -(pixdx*xwsize/(GLdouble) viewport[2] +
           eyedx*near/focus);
    dy = -(pixdy*ywsize/(GLdouble) viewport[3] +
           eyedy*near/focus);
   glMatrixMode(GL PROJECTION);
   glLoadIdentity();
    glFrustum (left + dx, right + dx, bottom + dy, top + dy,
       near, far);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
    glTranslatef (-eyedx, -eyedy, 0.0);
}
void accPerspective(GLdouble fovy, GLdouble aspect,
    GLdouble near, GLdouble far, GLdouble pixdx, GLdouble pixdy,
    GLdouble eyedx, GLdouble eyedy, GLdouble focus)
{
   GLdouble fov2, left, right, bottom, top;
   fov2 = ((fovy*PI) / 180.0) / 2.0;
   top = near / (fcos(fov2) / fsin(fov2));
   bottom = -top;
   right = top * aspect;
   left = -right;
    accFrustum (left, right, bottom, top, near, far,
       pixdx, pixdy, eyedx, eyedy, focus);
}
```

Example 10-3 uses these two routines to perform scene antialiasing.

Example 10-3 : Scene Antialiasing: accpersp.c

#include <GL/gl.h>
#include <GL/glu.h>

```
#include <stdlib.h>
#include <math.h>
#include <GL/glut.h>
#include "jitter.h"
void init(void)
{
  GLfloat mat_ambient[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat light_position[] = { 0.0, 0.0, 10.0, 1.0 };
  GLfloat lm_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
  glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialf(GL_FRONT, GL_SHININESS, 50.0);
  glLightfv(GL_LIGHT0, GL_POSITION, light_position);
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lm_ambient);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_DEPTH_TEST);
  glShadeModel (GL FLAT);
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClearAccum(0.0, 0.0, 0.0, 0.0);
}
void displayObjects(void)
{
  GLfloat torus_diffuse[] = { 0.7, 0.7, 0.0, 1.0 };
  GLfloat cube_diffuse[] = { 0.0, 0.7, 0.7, 1.0 };
   GLfloat sphere_diffuse[] = { 0.7, 0.0, 0.7, 1.0 };
  GLfloat octa_diffuse[] = { 0.7, 0.4, 0.4, 1.0 };
  glPushMatrix ();
  glTranslatef (0.0, 0.0, -5.0);
  glRotatef (30.0, 1.0, 0.0, 0.0);
  glPushMatrix ();
  glTranslatef (-0.80, 0.35, 0.0);
  glRotatef (100.0, 1.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, torus_diffuse);
   glutSolidTorus (0.275, 0.85, 16, 16);
  glPopMatrix ();
   glPushMatrix ();
   glTranslatef (-0.75, -0.50, 0.0);
   glRotatef (45.0, 0.0, 0.0, 1.0);
   glRotatef (45.0, 1.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, cube_diffuse);
   glutSolidCube (1.5);
  glPopMatrix ();
   glPushMatrix ();
   glTranslatef (0.75, 0.60, 0.0);
   glRotatef (30.0, 1.0, 0.0, 0.0);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, sphere_diffuse);
  glutSolidSphere (1.0, 16, 16);
  glPopMatrix ();
  glPushMatrix ();
  glTranslatef (0.70, -0.90, 0.25);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, octa_diffuse);
  glutSolidOctahedron ();
   glPopMatrix ();
```

```
glPopMatrix ();
}
#define ACSIZE
                8
void display(void)
   GLint viewport[4];
   int jitter;
   glGetIntegerv (GL_VIEWPORT, viewport);
   glClear(GL_ACCUM_BUFFER_BIT);
   for (jitter = 0; jitter < ACSIZE; jitter++) {</pre>
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      accPerspective (50.0,
         (GLdouble) viewport[2]/(GLdouble) viewport[3],
         1.0, 15.0, j8[jitter].x, j8[jitter].y, 0.0, 0.0, 1.0);
      displayObjects ();
      glAccum(GL_ACCUM, 1.0/ACSIZE);
   }
   glAccum (GL_RETURN, 1.0);
   glFlush();
}
void reshape(int w, int h)
ł
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}
/*
   Main Loop
  Be certain you request an accumulation buffer.
 * /
int main(int argc, char** argv)
ł
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                         | GLUT_ACCUM | GLUT_DEPTH);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

You don't have to use a perspective projection to perform scene antialiasing. You can antialias a scene with orthographic projection simply by using **glTranslate***() to jitter the scene. Keep in mind that **glTranslate***() operates in world coordinates, but you want the apparent motion of the scene to be less than one pixel, measured in screen coordinates. Thus, you must reverse the world-coordinate mapping by calculating the jittering translation values, using its width or height in world coordinates divided by its viewport size. Then multiply that world-coordinate value by the amount of jitter to determine how much the scene should be moved in world coordinates to get a predictable jitter of less than one pixel. Example 10-4 shows how the **display(**) and **reshape(**) routines might look with a world-coordinate width and height of 4.5.

Example 10-4 : Jittering with an Orthographic Projection: accanti.c

#define ACSIZE 8

```
void display(void)
   GLint viewport[4];
   int jitter;
   glGetIntegerv (GL_VIEWPORT, viewport);
   glClear(GL_ACCUM_BUFFER_BIT);
   for (jitter = 0; jitter < ACSIZE; jitter++) {</pre>
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glPushMatrix ();
/*
        Note that 4.5 is the distance in world space between
 *
        left and right and bottom and top.
 *
        This formula converts fractional pixel movement to
 *
        world coordinates.
 * /
      glTranslatef (j8[jitter].x*4.5/viewport[2],
                    j8[jitter].y*4.5/viewport[3], 0.0);
      displayObjects ();
      glPopMatrix ();
      glAccum(GL_ACCUM, 1.0/ACSIZE);
   glAccum (GL_RETURN, 1.0);
   glFlush();
}
void reshape(int w, int h)
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w \le h)
      glOrtho (-2.25, 2.25, -2.25*h/w, 2.25*h/w, -10.0, 10.0);
   else
      glOrtho (-2.25*w/h, 2.25*w/h, -2.25, 2.25, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
```

Motion Blur

Similar methods can be used to simulate motion blur, as shown in "Plate 7" in Appendix I and Figure 10-2. Suppose your scene has some stationary and some moving objects in it, and you want to make a motion-blurred image extending over a small interval of time. Set up the accumulation buffer in the same way, but instead of spatially jittering the images, jitter them temporally. The entire scene can be made successively dimmer by calling

```
glAccum (GL_MULT, decayFactor);
```

as the scene is drawn into the accumulation buffer, where *decayFactor* is a number from 0.0 to 1.0. Smaller numbers for *decayFactor* cause the object to appear to be moving faster. You can transfer the completed scene with the object's current position and "vapor trail" of previous positions from the accumulation buffer to the standard color buffer with

```
glAccum (GL_RETURN, 1.0);
```

The image looks correct even if the items move at different speeds, or if some of them are accelerated. As before, the more jitter points (temporal, in this case) you use, the better the final image, at least up to the point where you begin to lose resolution due to finite precision in the accumulation buffer. You can combine motion blur with antialiasing by jittering in both the spatial

and temporal domains, but you pay for higher quality with longer rendering times.

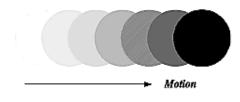


Figure 10-2 : Motion-Blurred Object

Depth of Field

A photograph made with a camera is in perfect focus only for items lying on a single plane a certain distance from the film. The farther an item is from this plane, the more out of focus it is. The depth of field for a camera is a region about the plane of perfect focus where items are out of focus by a small enough amount.

Under normal conditions, everything you draw with OpenGL is in focus (unless your monitor's bad, in which case everything is out of focus). The accumulation buffer can be used to approximate what you would see in a photograph where items are more and more blurred as their distance from a plane of perfect focus increases. It isn't an exact simulation of the effects produced in a camera, but the result looks similar to what a camera would produce.

To achieve this result, draw the scene repeatedly using calls with different argument values to **glFrustum**(). Choose the arguments so that the position of the viewpoint varies slightly around its true position and so that each frustum shares a common rectangle that lies in the plane of perfect focus, as shown in Figure 10-3. The results of all the renderings should be averaged in the usual way using the accumulation buffer.

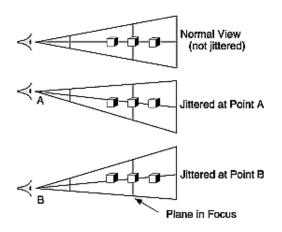


Figure 10-3 : Jittered Viewing Volume for Depth-of-Field Effects

"Plate 10" in Appendix I shows an image of five teapots drawn using the depth-of-field effect. The gold teapot (second from the left) is in focus, and the other teapots get progressively blurrier, depending upon their distance from the focal plane (gold teapot). The code to draw this image is shown in Example 10-5 (which assumes **accPerspective(**) and **accFrustum(**) are defined as

described in Example 10-2). The scene is drawn eight times, each with a slightly jittered viewing volume, by calling **accPerspective(**). As you recall, with scene antialiasing, the fifth and sixth parameters jitter the viewing volumes in the *x* and *y* directions. For the depth-of-field effect, however, you want to jitter the volume while holding it stationary at the focal plane. The focal plane is the depth value defined by the ninth (last) parameter to **accPerspective(**), which is z = 5.0 in this example. The amount of blur is determined by multiplying the *x* and *y* jitter values (seventh and eighth parameters of **accPerspective(**)) by a constant. Determining the constant is not a science; experiment with values until the depth of field is as pronounced as you want. (Note that in Example 10-5, the fifth and sixth parameters to **accPerspective(**) are set to 0.0, so scene antialiasing is turned off.)

Example 10-5 : Depth-of-Field Effect: dof.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>
#include "jitter.h"
void init(void)
   GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };
   GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
   GLfloat local_view[] = { 0.0 };
   glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
   glLightfv(GL_LIGHT0, GL_POSITION, position);
   glLightModelfv(GL LIGHT MODEL AMBIENT, lmodel ambient);
   glLightModelfv(GL LIGHT MODEL LOCAL VIEWER, local view);
   glFrontFace (GL CW);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);
   glEnable(GL_DEPTH_TEST);
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClearAccum(0.0, 0.0, 0.0, 0.0);
/* make teapot display list */
   teapotList = glGenLists(1);
   glNewList (teapotList, GL_COMPILE);
   glutSolidTeapot (0.5);
   glEndList ();
}
void renderTeapot (GLfloat x, GLfloat y, GLfloat z,
   GLfloat ambr, GLfloat ambg, GLfloat ambb,
   GLfloat difr, GLfloat difg, GLfloat difb,
   GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine)
{
   GLfloat mat[4];
   glPushMatrix();
```

```
glTranslatef (x, y, z);
   mat[0] = ambr; mat[1] = ambq; mat[2] = ambb; mat[3] = 1.0;
   glMaterialfv (GL_FRONT, GL_AMBIENT, mat);
   mat[0] = difr; mat[1] = difg; mat[2] = difb;
   glMaterialfv (GL_FRONT, GL_DIFFUSE, mat);
   mat[0] = specr; mat[1] = specg; mat[2] = specb;
   glMaterialfv (GL_FRONT, GL_SPECULAR, mat);
   glMaterialf (GL_FRONT, GL_SHININESS, shine*128.0);
   glCallList(teapotList);
   glPopMatrix();
}
void display(void)
   int jitter;
  GLint viewport[4];
   glGetIntegerv (GL_VIEWPORT, viewport);
   glClear(GL_ACCUM_BUFFER_BIT);
   for (jitter = 0; jitter < 8; jitter++) {</pre>
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      accPerspective (45.0,
         (GLdouble) viewport[2]/(GLdouble) viewport[3],
         1.0, 15.0, 0.0, 0.0,
         0.33*j8[jitter].x, 0.33*j8[jitter].y, 5.0);
/*
        ruby, gold, silver, emerald, and cyan teapots
                                                         * /
      renderTeapot (-1.1, -0.5, -4.5, 0.1745, 0.01175,
                    0.01175, 0.61424, 0.04136, 0.04136,
                    0.727811, 0.626959, 0.626959, 0.6);
      renderTeapot (-0.5, -0.5, -5.0, 0.24725, 0.1995,
                    0.0745, 0.75164, 0.60648, 0.22648,
                    0.628281, 0.555802, 0.366065, 0.4);
      renderTeapot (0.2, -0.5, -5.5, 0.19225, 0.19225,
                    0.19225, 0.50754, 0.50754, 0.50754,
                    0.508273, 0.508273, 0.508273, 0.4);
      renderTeapot (1.0, -0.5, -6.0, 0.0215, 0.1745, 0.0215,
                    0.07568, 0.61424, 0.07568, 0.633,
                    0.727811, 0.633, 0.6);
      renderTeapot (1.8, -0.5, -6.5, 0.0, 0.1, 0.06, 0.0,
                    0.50980392, 0.50980392, 0.50196078,
                    0.50196078, 0.50196078, .25);
      glAccum (GL_ACCUM, 0.125);
   }
   glAccum (GL_RETURN, 1.0);
   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
}
/* Main Loop
 * Be certain you request an accumulation buffer.
 */
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB
                         GLUT_ACCUM | GLUT_DEPTH);
  glutInitWindowSize (400, 400);
  glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
```

```
init();
glutReshapeFunc(reshape);
glutDisplayFunc(display);
glutMainLoop();
return 0;
```

Soft Shadows

}

To accumulate soft shadows due to multiple light sources, render the shadows with one light turned on at a time, and accumulate them together. This can be combined with spatial jittering to antialias the scene at the same time. (See "Shadows" in Chapter 14 for more information about drawing shadows.)

Jittering

If you need to take nine or sixteen samples to antialias an image, you might think that the best choice of points is an equally spaced grid across the pixel. Surprisingly, this is not necessarily true. In fact, sometimes it's a good idea to take points that lie in adjacent pixels. You might want a uniform distribution or a normalized distribution, clustering toward the center of the pixel. (The aforementioned SIGGRAPH paper discusses these issues.) In addition, Table 10-5 shows a few sets of reasonable jittering values to be used for some selected sample counts. Most of the examples in the table are uniformly distributed in the pixel, and all lie within the pixel.

Count	Values
2	$\{0.25, 0.75\}, \{0.75, 0.25\}$
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877},
	{0.2261828938, 0.4131553612}
4	$\{0.375, 0.25\}, \{0.125, 0.75\}, \{0.875, 0.25\}, \{0.625, 0.75\}$
5	$\{0.5, 0.5\}, \{0.3, 0.1\}, \{0.7, 0.9\}, \{0.9, 0.3\}, \{0.1, 0.7\}$
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979},
	$\{0.5353535353, 0.8686868686\}, \{0.8686868686, 0.5353535353\},\$
	{0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	$\{0.5625, 0.4375\}, \{0.0625, 0.9375\}, \{0.3125, 0.6875\}, \{0.6875, 0.8125\}, \{0.8125, 0.1875\}, \{0.9375, 0.5625\}, \{0.4375, 0.0625\}, \{0.1875, 0.3125\}$

9	$\{0.5, 0.5\}, \{0.16666666666, 0.944444444\}, \{0.5, 0.166666666666\},$
	$\{0.5, 0.8333333333\}, \{0.16666666666, 0.27777777777\},$
	$\{0.833333333, 0.3888888888\}, \{0.166666666666, 0.611111111\},$
	$\{0.833333333, 0.7222222222\}, \{0.8333333333, 0.0555555555\}$
12	{0.41666666666, 0.625}, {0.91666666666, 0.875}, {0.25, 0.375},
	$\{0.41666666666, 0.125\}, \{0.75, 0.125\}, \{0.0833333333, 0.125\}, \{0.75, 0.625\},$
	$\{0.25, 0.875\}, \{0.5833333333, 0.375\}, \{0.91666666666, 0.375\},$
	$\{0.0833333333, 0.625\}, \{0.583333333, 0.875\}$
16	$\{0.375, 0.4375\}, \{0.625, 0.0625\}, \{0.875, 0.1875\}, \{0.125, 0.0625\},$
	$\{0.375, 0.6875\}, \{0.875, 0.4375\}, \{0.625, 0.5625\}, \{0.375, 0.9375\},$
	$\{0.625, 0.3125\}, \{0.125, 0.5625\}, \{0.125, 0.8125\}, \{0.375, 0.1875\},$
	$\{0.875, 0.9375\}, \{0.875, 0.6875\}, \{0.125, 0.3125\}, \{0.625, 0.8125\}$

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 11 Tessellators and Quadrics

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Render concave filled polygons by first tessellating them into convex polygons, which can be rendered using standard OpenGL routines.
- Use the GLU library to create quadrics objects to render and model the surfaces of spheres and cylinders and to tessellate disks (circles) and partial disks (arcs).

The OpenGL library (GL) is designed for low-level operations, both streamlined and accessible to hardware acceleration. The OpenGL Utility Library (GLU) complements the OpenGL library, supporting higher-level operations. Some of the GLU operations are covered in other chapters. Mipmapping (gluBuild*DMipmaps()) and image scaling (gluScaleImage()) are discussed along with other facets of texture mapping in Chapter 9. Several matrix transformation GLU routines (gluOrtho2D(), gluPerspective(), gluLookAt(), gluProject(), and gluUnProject()) are described in Chapter 3. The use of gluPickMatrix() is explained in Chapter 13. The GLU NURBS facilities, which are built atop OpenGL evaluators, are covered in Chapter 12. Only two GLU topics remain: polygon tessellators and quadric surfaces, and those topics are discussed in this chapter.

To optimize performance, the basic OpenGL only renders convex polygons, but the GLU contains routines to tessellate concave polygons into convex ones, which the basic OpenGL can handle. Where the basic OpenGL operates upon simple primitives, such as points, lines, and filled polygons, the GLU can create higher-level objects, such as the surfaces of spheres, cylinders, and cones.

This chapter has the following major sections.

- "Polygon Tessellation" explains how to tessellate convex polygons into easier-to-render convex polygons.
- "Quadrics: Rendering Spheres, Cylinders, and Disks" describes how to generate spheres, cylinders, circles and arcs, including data such as surface normals and texture coordinates.

Polygon Tessellation

As discussed in "Describing Points, Lines, and Polygons" in Chapter 2, OpenGL can directly display only simple convex polygons. A polygon is simple if the edges intersect only at vertices,

there are no duplicate vertices, and exactly two edges meet at any vertex. If your application requires the display of concave polygons, polygons containing holes, or polygons with intersecting edges, those polygons must first be subdivided into simple convex polygons before they can be displayed. Such subdivision is called tessellation, and the GLU provides a collection of routines that perform tessellation. These routines take as input arbitrary contours, which describe hard-to-render polygons, and they return some combination of triangles, triangle meshes, triangle fans, or lines.

Figure 11-1 shows some contours of polygons that require tessellation: from left to right, a concave polygon, a polygon with a hole, and a self-intersecting polygon.

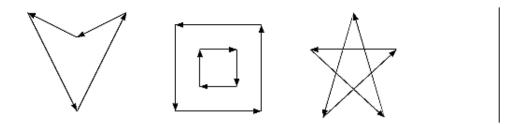


Figure 11-1 : Contours That Require Tessellation

If you think a polygon may need tessellation, follow these typical steps.

- 1. Create a new tessellation object with gluNewTess().
- 2. Use **gluTessCallback**() several times to register callback functions to perform operations during the tessellation. The trickiest case for a callback function is when the tessellation algorithm detects an intersection and must call the function registered for the GLU_TESS_COMBINE callback.
- 3. Specify tessellation properties by calling **gluTessProperty**(). The most important property is the winding rule, which determines the regions that should be filled and those that should remain unshaded.
- 4. Create and render tessellated polygons by specifying the contours of one or more closed polygons. If the data for the object is static, encapsulate the tessellated polygons in a display list. (If you don't have to recalculate the tessellation over and over again, using display lists is more efficient.)
- 5. If you need to tessellate something else, you may reuse your tessellation object. If you are forever finished with your tessellation object, you may delete it with **gluDeleteTess**().

Note: The tessellator described here was introduced in version 1.2 of the GLU. If you are using an older version of the GLU, you must use routines described in "Describing GLU Errors". To query which version of GLU you have, use **gluGetString**(**GLU_VERSION**), which returns a string with your GLU version number. If you don't seem to have **gluGetString**() in your GLU, then you have GLU 1.0, which did not yet have the **gluGetString**() routine.

Create a Tessellation Object

As a complex polygon is being described and tessellated, it has associated data, such as the vertices, edges, and callback functions. All this data is tied to a single tessellation object. To perform tessellation, your program first has to create a tessellation object using the routine **gluNewTess**().

GLUtesselator* gluNewTess(void);

Creates a new tessellation object and returns a pointer to it. A null pointer is returned if the creation fails.

A single tessellation object can be reused for all your tessellations. This object is required only because library routines might need to do their own tessellations, and they should be able to do so without interfering with any tessellation that your program is doing. It might also be useful to have multiple tessellation objects if you want to use different sets of callbacks for different tessellations. A typical program, however, allocates a single tessellation object and uses it for all its tessellations. There's no real need to free it because it uses a small amount of memory. On the other hand, it never hurts to be tidy.

Tessellation Callback Routines

After you create a tessellation object, you must provide a series of callback routines to be called at appropriate times during the tessellation. After specifying the callbacks, you describe the contours of one or more polygons using GLU routines. When the description of the contours is complete, the tessellation facility invokes your callback routines as necessary.

Any functions that are omitted are simply not called during the tessellation, and any information they might have returned to your program is lost. All are specified by the single routine **gluTessCallback**().

void **gluTessCallback**(GLUtesselator *tessobj, GLenum type, void (*fn)()); Associates the callback function fn with the tessellation object tessobj. The type of the callback is determined by the parameter type, which can be GLU TESS BEGIN, GLU_TESS_BEGIN_DATA, GLU_TESS_EDGE_FLAG, GLU_TESS_EDGE_FLAG_DATA, GLU TESS VERTEX, GLU TESS VERTEX DATA, GLU TESS END, GLU_TESS_END_DATA, GLU_TESS_COMBINE, GLU_TESS_COMBINE_DATA, GLU_TESS_ERROR, and GLU_TESS_ERROR_DATA. The twelve possible callback functions have the following prototypes: GLU_TESS_BEGIN void begin(GLenum type); GLU_TESS_BEGIN_DATA void begin(GLenum type, *void* **user data*); GLU_TESS_EDGE_FLAG void edgeFlag(GLboolean flag); GLU_TESS_EDGE_FLAG_DATA void edgeFlag(GLboolean flag, void *user data); GLU TESS VERTEX void vertex(void *vertex data); GLU TESS VERTEX DATA void vertex (void *vertex data, void *user data); GLU_TESS_END void end(void); GLU TESS END DATA void end(void *user data); GLU TESS ERROR void error(GLenum errno); GLU_TESS_ERROR_DATA void error(GLenum errno, void *user_data); GLU_TESS_COMBINE void combine(GLdouble coords[3], void *vertex_data[4], *GLfloat weight*[4],

void **outData); GLU_TESS_COMBINE_DATA void combine(GLdouble coords[3], void *vertex_data[4], GLfloat weight[4], void **outData, void *user_data);

To change a callback routine, simply call **gluTessCallback()** with the new routine. To eliminate a callback routine without replacing it with a new one, pass **gluTessCallback()** a null pointer for the appropriate function.

As tessellation proceeds, the callback routines are called in a manner similar to how you use the OpenGL commands **glBegin()**, **glEdgeFlag*()**, **glVertex*()**, and **glEnd()**. (See "Marking Polygon Boundary Edges" in Chapter 2 for more information about **glEdgeFlag*()**.) The combine callback is used to create new vertices where edges intersect. The error callback is invoked during the tessellation only if something goes wrong.

For every tessellator object created, a GLU_TESS_BEGIN callback is invoked with one of four possible parameters: GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, GL_TRIANGLES, and GL_LINE_LOOP. When the tessellator decomposes the polygons, the tessellation algorithm will decide which type of triangle primitive is most efficient to use. (If the GLU_TESS_BOUNDARY_ONLY property is enabled, then GL_LINE_LOOP is used for rendering.)

Since edge flags make no sense in a triangle fan or triangle strip, if there is a callback associated with GLU_TESS_EDGE_FLAG that enables edge flags, the GLU_TESS_BEGIN callback is called only with GL_TRIANGLES. The GLU_TESS_EDGE_FLAG callback works exactly analogously to the OpenGL **glEdgeFlag***() call.

After the GLU_TESS_BEGIN callback routine is called and before the callback associated with GLU_TESS_END is called, some combination of the GLU_TESS_EDGE_FLAG and GLU_TESS_VERTEX callbacks is invoked (usually by calls to **gluTessVertex(**), which is described on page 425). The associated edge flags and vertices are interpreted exactly as they are in OpenGL between **glBegin(**) and the matching **glEnd(**).

If something goes wrong, the error callback is passed a GLU error number. A character string describing the error is obtained using the routine **gluErrorString()**. (See "Describing GLU Errors" for more information about this routine.)

Example 11-1 shows a portion of tess.c, where a tessellation object is created and several callbacks are registered.

Example 11-1 : Registering Tessellation Callbacks: tess.c

```
(GLvoid (*) ()) & errorCallback);
  the callback routines registered by gluTessCallback() */
/*
void beginCallback(GLenum which)
{
   glBegin(which);
}
void endCallback(void)
{
   glEnd();
}
void errorCallback(GLenum errorCode)
   const GLubyte *estring;
   estring = gluErrorString(errorCode);
   fprintf (stderr, "Tessellation Error: %s\n", estring);
   exit (0);
}
```

In Example 11-1, the registered GLU_TESS_VERTEX callback is simply **glVertex3dv**(), and only the coordinates at each vertex are passed along. However, if you want to specify more information at every vertex, such as a color value, a surface normal vector, or texture coordinate, you'll have to make a more complex callback routine. Example 11-2 shows the start of another tessellated object, further along in program tess.c. The registered function **vertexCallback**() expects to receive a parameter that is a pointer to six double-length floating point values: the *x*, *y*, and *z* coordinates and the red, green, and blue color values, respectively, for that vertex.

Example 11-2 : Vertex and Combine Callbacks: tess.c

```
/* a different portion of init() */
   gluTessCallback(tobj, GLU_TESS_VERTEX,
                   (GLvoid (*) ()) &vertexCallback);
   gluTessCallback(tobj, GLU_TESS_BEGIN,
                   (GLvoid (*) ()) &beginCallback);
   gluTessCallback(tobj, GLU_TESS_END,
                   (GLvoid (*) ()) &endCallback);
   gluTessCallback(tobj, GLU_TESS_ERROR,
                   (GLvoid (*) ()) & errorCallback);
   gluTessCallback(tobj, GLU_TESS_COMBINE,
                   (GLvoid (*) ()) & combineCallback);
/* new callback routines registered by these calls */
void vertexCallback(GLvoid *vertex)
   const GLdouble *pointer;
   pointer = (GLdouble *) vertex;
   glColor3dv(pointer+3);
   glVertex3dv(vertex);
}
void combineCallback(GLdouble coords[3],
                     GLdouble *vertex_data[4],
                     GLfloat weight[4], GLdouble **dataOut )
{
   GLdouble *vertex;
   int i;
```

Example 11-2 also shows the use of the GLU_TESS_COMBINE callback. Whenever the tessellation algorithm examines the input contours, detects an intersection, and decides it must create a new vertex, the GLU_TESS_COMBINE callback is invoked. The callback is also called when the tessellator decides to merge features of two vertices that are very close to one another. The newly created vertex is a linear combination of up to four existing vertices, referenced by *vertex_data*[0..3] in Example 11-2. The coefficients of the linear combination are given by *weight*[0..3]; these weights sum to 1.0. *coords* gives the location of the new vertex.

The registered callback routine must allocate memory for another vertex, perform a weighted interpolation of data using *vertex_data* and *weight*, and return the new vertex pointer as *dataOut*. **combineCallback()** in Example 11-2 interpolates the RGB color value. The function allocates a six-element array, puts the *x*, *y*, and *z* coordinates in the first three elements, and then puts the weighted average of the RGB color values in the last three elements.

User-Specified Data

}

Six kinds of callbacks can be registered. Since there are two versions of each kind of callback, there are twelve callbacks in all. For each kind of callback, there is one with user-specified data and one without. The user-specified data is given by the application to **gluTessBeginPolygon**() and is then passed, unaltered, to each *DATA callback routine. With GLU_TESS_BEGIN_DATA, the user-specified data may be used for "per-polygon" data. If you specify both versions of a particular callback, the callback with *user_data* is used, and the other is ignored. So, although there are twelve callbacks, you can have a maximum of six callback functions active at any time.

For instance, Example 11-2 uses smooth shading, so **vertexCallback()** specifies an RGB color for every vertex. If you want to do lighting and smooth shading, the callback would specify a surface normal for every vertex. However, if you want lighting and flat shading, you might specify only one surface normal for every polygon, not for every vertex. In that case, you might choose to use the GLU_TESS_BEGIN_DATA callback and pass the vertex coordinates and surface normal in the *user_data* pointer.

Tessellation Properties

Prior to tessellation and rendering, you may use **gluTessProperty**() to set several properties to affect the tessellation algorithm. The most important and complicated of these properties is the winding rule, which determines what is considered "interior" and "exterior."

void **gluTessProperty**(*GLUtesselator* **tessobj*, *GLenum property*, *GLdouble value*);

For the tessellation object tessobj, the current value of property is set to value. property is one of GLU_TESS_BOUNDARY_ONLY, GLU_TESS_TOLERANCE, or

GLU_TESS_WINDING_RULE.

If property is GLU_TESS_BOUNDARY_ONLY, value is either GL_TRUE or GL_FALSE. When set to GL_TRUE, polygons are no longer tessellated into filled polygons; line loops are drawn to outline the contours that separate the polygon interior and exterior. The default value is GL_FALSE. (See **gluTessNormal**() to see how to control the winding direction of the contours.)

If property is GLU_TESS_TOLERANCE, value is a distance used to calculate whether two vertices are close together enough to be merged by the GLU_TESS_COMBINE callback. The tolerance value is multiplied by the largest coordinate magnitude of an input vertex to determine the maximum distance any feature can move as a result of a single merge operation. Feature merging may not be supported by your implementation, and the tolerance value is only a hint. The default tolerance value is zero.

The GLU_TESS_WINDING_RULE property determines which parts of the polygon are on the interior and which are the exterior and should not be filled. value can be one of GLU_TESS_WINDING_ODD (the default), GLU_TESS_WINDING_NONZERO, GLU_TESS_WINDING_POSITIVE, GLU_TESS_WINDING_NEGATIVE, or GLU_TESS_WINDING_ABS_GEQ_TWO.

Winding Numbers and Winding Rules

For a single contour, the winding number of a point is the signed number of revolutions we make around that point while traveling once around the contour (where a counterclockwise revolution is positive and a clockwise revolution is negative). When there are several contours, the individual winding numbers are summed. This procedure associates a signed integer value with each point in the plane. Note that the winding number is the same for all points in a single region.

Figure 11-2 shows three sets of contours and winding numbers for points inside those contours. In the left set, all three contours are counterclockwise, so each nested interior region adds one to the winding number. For the middle set, the two interior contours are drawn clockwise, so the winding number decreases and actually becomes negative.

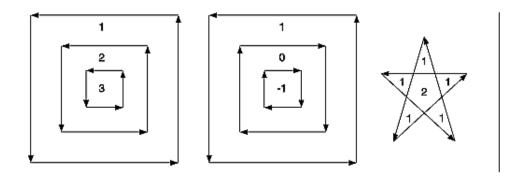


Figure 11-2 : Winding Numbers for Sample Contours

The winding rule classifies a region as *inside* if its winding number belongs to the chosen category (odd, nonzero, positive, negative, or "absolute value of greater than or equal to two"). The odd and nonzero rules are common ways to define the interior. The positive, negative, and "absolute value>=2" winding rules have some limited use for polygon CSG (computational solid geometry) operations.

The program tesswind.c demonstrates the effects of winding rules. The four sets of contours shown in Figure 11-3 are rendered. The user can then cycle through the different winding rule properties to see their effects. For each winding rule, the dark areas represent interiors. Note the effect of clockwise and counterclockwise winding.

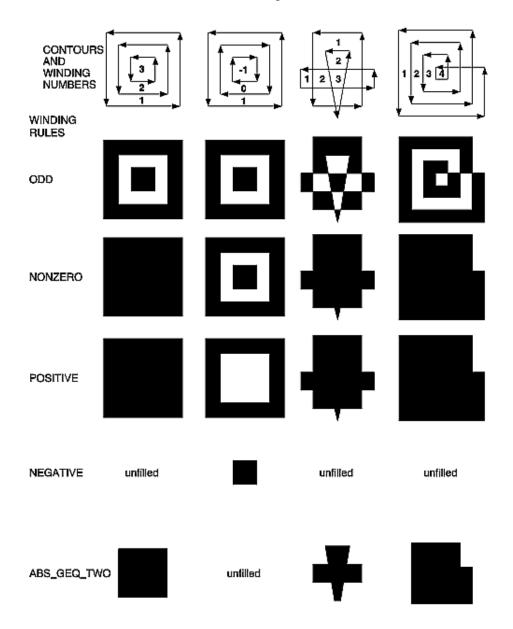


Figure 11-3 : How Winding Rules Define Interiors

CSG Uses for Winding Rules

GLU_TESS_WINDING_ODD and GLU_TESS_WINDING_NONZERO are the most commonly used winding rules. They work for the most typical cases of shading.

The winding rules are also designed for computational solid geometry (CSG) operations. Thy make it easy to find the union, difference, or intersection (Boolean operations) of several contours.

First, assume that each contour is defined so that the winding number is zero for each exterior

region and one for each interior region. (Each contour must not intersect itself.) Under this model, counterclockwise contours define the outer boundary of the polygon, and clockwise contours define holes. Contours may be nested, but a nested contour must be oriented oppositely from the contour that contains it.

If the original polygons do not satisfy this description, they can be converted to this form by first running the tessellator with the GLU_TESS_BOUNDARY_ONLY property turned on. This returns a list of contours satisfying the restriction just described. By creating two tessellator objects, the callbacks from one tessellator can be fed directly as input to the other.

Given two or more polygons of the preceding form, CSG operations can be implemented as follows.

- UNION To calculate the union of several contours, draw all input contours as a single polygon. The winding number of each resulting region is the number of original polygons that cover it. The union can be extracted by using the GLU_TESS_WINDING_NONZERO or GLU_TESS_WINDING_POSITIVE winding rules. Note that with the nonzero winding rule, we would get the same result if all contour orientations were reversed.
- INTERSECTION This only works for two contours at a time. Draw a single polygon using two contours. Extract the result using GLU_TESS_WINDING_ABS_GEQ_TWO.
- DIFFERENCE Suppose you want to compute A diff (B union C union D). Draw a single polygon consisting of the unmodified contours from A, followed by the contours of B, C, and D, with their vertex order reversed. To extract the result, use the GLU_TESS_WINDING_POSITIVE winding rule. (If B, C, and D are the result of a GLU_TESS_BOUNDARY_ONLY operation, an alternative to reversing the vertex order is to use **gluTessNormal**() to reverse the sign of the supplied normal.

Other Tessellation Property Routines

There are complementary routines, which work alongside **gluTessProperty**(). **gluGetTessProperty**() retrieves the current values of tessellator properties. If the tessellator is being used to generate wire frame outlines instead of filled polygons, **gluTessNormal**() can be used to determine the winding direction of the tessellated polygons.

void **gluGetTessProperty**(GLUtesselator *tessobj, GLenum property, GLdouble *value);

For the tessellation object tessobj, the current value of property is returned to value. Values for property and value are the same as for **gluTessProperty**().

void **gluTessNormal**(GLUtesselator *tessobj, GLdouble x, GLdouble y, GLdouble z);

For the tessellation object tessobj, **gluTessNormal()** defines a normal vector, which controls the winding direction of generated polygons. Before tessellation, all input data is projected into a plane perpendicular to the normal. Then, all output triangles are oriented counterclockwise, with respect to the normal. (Clockwise orientation can be obtained by reversing the sign of the supplied normal.) The default normal is (0, 0, 0).

If you have some knowledge about the location and orientation of the input data, then using **gluTessNormal**() can increase the speed of the tessellation. For example, if you know that all polygons lie on the x-y plane, call **gluTessNormal**(*tessobj*, 0, 0, 1).

The default normal is (0, 0, 0), and its effect is not immediately obvious. In this case, it is expected that the input data lies approximately in a plane, and a plane is fitted to the vertices, no matter how they are truly connected. The sign of the normal is chosen so that the sum of the signed areas of all input contours is nonnegative (where a counterclockwise contour has a positive area). Note that if the input data does not lie approximately in a plane, then projection perpendicular to the computed normal may substantially change the geometry.

Polygon Definition

After all the tessellation properties have been set and the callback actions have been registered, it is finally time to describe the vertices that compromise input contours and tessellate the polygons.

void gluTessBeginPolygon (GLUtesselator *tessobj, void *user_data); void gluTessEndPolygon (GLUtesselator *tessobj);

Begins and ends the specification of a polygon to be tessellated and associates a tessellation object, tessobj, with it. user_data points to a user-defined data structure, which is passed along all the GLU_TESS_*_DATA callback functions that have been bound.

Calls to **gluTessBeginPolygon**() and **gluTessEndPolygon**() surround the definition of one or more contours. When **gluTessEndPolygon**() is called, the tessellation algorithm is implemented, and the tessellated polygons are generated and rendered. The callback functions and tessellation properties that were bound and set to the tessellation object using **gluTessCallback**() and **gluTessProperty**() are used.

void gluTessBeginContour (GLUtesselator *tessobj);

void gluTessEndContour (GLUtesselator *tessobj);

Begins and ends the specification of a closed contour, which is a portion of a polygon. A closed contour consists of zero or more calls to **gluTessVertex()**, which defines the vertices. The last vertex of each contour is automatically linked to the first.

In practice, a minimum of three vertices is needed for a meaningful contour.

void gluTessVertex (GLUtesselator *tessobj, GLdouble coords[3],

void **vertex_data*);

Specifies a vertex in the current contour for the tessellation object. coords contains the three-dimensional vertex coordinates, and vertex_data is a pointer that's sent to the callback associated with GLU_TESS_VERTEX or GLU_TESS_VERTEX_DATA. Typically, vertex_data contains vertex coordinates, surface normals, texture coordinates, color information, or whatever else the application may find useful.

In the program tess.c, a portion of which is shown in Example 11-3, two polygons are defined. One polygon is a rectangular contour with a triangular hole inside, and the other is a smooth-shaded, self-intersecting, five-pointed star. For efficiency, both polygons are stored in display lists. The first polygon consists of two contours; the outer one is wound counterclockwise, and the "hole" is wound clockwise. For the second polygon, the *star* array contains both the coordinate and color data, and its tessellation callback, **vertexCallback(**), uses both.

It is important that each vertex is in a different memory location because the vertex data is not copied by **gluTessVertex()**; only the pointer (*vertex_data*) is saved. A program that reuses the same memory for several vertices may not get the desired result.

Note: In **gluTessVertex**(), it may seem redundant to specify the vertex coordinate data twice, for both the *coords* and *vertex_data* parameters; however, both are necessary. *coords* refers only to the vertex coordinates. *vertex_data* uses the coordinate data, but may also use other information for each vertex.

Example 11-3 : Polygon Definition: tess.c

```
GLdouble rect[4][3] = \{50.0, 50.0, 0.0, 
                        200.0, 50.0, 0.0,
                        200.0, 200.0, 0.0,
50.0, 200.0, 0.0};
GLdouble tri[3][3] = {75.0, 75.0, 0.0,
                       125.0, 175.0, 0.0,
                       175.0, 75.0, 0.0};
GLdouble star[5][6] = {250.0, 50.0, 0.0, 1.0, 0.0, 1.0,
                        325.0, 200.0, 0.0, 1.0, 1.0, 0.0,
                        400.0, 50.0, 0.0, 0.0, 1.0, 1.0,
                        250.0, 150.0, 0.0, 1.0, 0.0, 0.0,
                        400.0, 150.0, 0.0, 0.0, 1.0, 0.0};
startList = glGenLists(2);
tobj = qluNewTess();
gluTessCallback(tobj, GLU_TESS_VERTEX,
                 (GLvoid (*) ()) &glVertex3dv);
gluTessCallback(tobj, GLU_TESS_BEGIN,
                 (GLvoid (*) ()) &beginCallback);
gluTessCallback(tobj, GLU_TESS_END,
                 (GLvoid (*) ()) & endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR,
                 (GLvoid (*) ()) & errorCallback);
glNewList(startList, GL_COMPILE);
glShadeModel(GL_FLAT);
gluTessBeginPolygon(tobj, NULL);
   gluTessBeginContour(tobj);
      gluTessVertex(tobj, rect[0], rect[0]);
      gluTessVertex(tobj, rect[1], rect[1]);
      gluTessVertex(tobj, rect[2], rect[2]);
      gluTessVertex(tobj, rect[3], rect[3]);
   gluTessEndContour(tobj);
   gluTessBeginContour(tobj);
      gluTessVertex(tobj, tri[0], tri[0]);
      gluTessVertex(tobj, tri[1], tri[1]);
      gluTessVertex(tobj, tri[2], tri[2]);
   gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();
gluTessCallback(tobj, GLU_TESS_VERTEX,
                 (GLvoid (*) ()) &vertexCallback);
gluTessCallback(tobj, GLU_TESS_BEGIN,
                 (GLvoid (*) ()) &beginCallback);
gluTessCallback(tobj, GLU_TESS_END,
                 (GLvoid (*) ()) & endCallback);
gluTessCallback(tobj, GLU_TESS_ERROR,
                 (GLvoid (*) ()) &errorCallback);
gluTessCallback(tobj, GLU_TESS_COMBINE,
                 (GLvoid (*) ()) &combineCallback);
glNewList(startList + 1, GL_COMPILE);
glShadeModel(GL_SMOOTH);
gluTessProperty(tobj, GLU_TESS_WINDING_RULE,
                GLU_TESS_WINDING_POSITIVE);
```

```
gluTessBeginPolygon(tobj, NULL);
gluTessBeginContour(tobj);
gluTessVertex(tobj, star[0], star[0]);
gluTessVertex(tobj, star[1], star[1]);
gluTessVertex(tobj, star[2], star[2]);
gluTessVertex(tobj, star[3], star[3]);
gluTessVertex(tobj, star[4], star[4]);
gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList();
```

Deleting a Tessellator Object

If you no longer need a tessellation object, you can delete it and free all associated memory with **gluDeleteTess**().

void **gluDeleteTess**(GLUtesselator *tessobj); Deletes the specified tessellation object, tessobj, and frees all associated memory.

Tessellator Performance Tips

For best performance, remember these rules.

- 1. Cache the output of the tessellator in a display list or other user structure. To obtain the post-tessellation vertex coordinates, tessellate the polygons while in feedback mode. (See "Feedback" in Chapter 13.)
- 2. Use **gluTessNormal**() to supply the polygon normal.
- 3. Use the same tessellator object to render many polygons rather than allocate a new tessellator for each one. (In a multithreaded, multiprocessor environment, you may get better performance using several tessellators.)

Describing GLU Errors

The GLU provides a routine for obtaining a descriptive string for an error code. This routine is not limited to tessellation but is also used for NURBS and quadrics errors, as well as errors in the base GL. (See "Error Handling" in Chapter 14 for information about OpenGL's error handling facility.)

Backward Compatibility

If you are using the 1.0 or 1.1 version of GLU, you have a much less powerful tessellator available. The 1.0/1.1 tessellator handles only simple nonconvex polygons or simple polygons containing holes. It does not properly tessellate intersecting contours (no COMBINE callback), nor process per-polygon data.

The 1.0/1.1 tessellator has some similarities to the current tessellator. **gluNewTess()** and **gluDeleteTess()** are used for both tessellators. The main vertex specification routine remains **gluTessVertex()**. The callback mechanism is controlled by **gluTessCallback()**, although there are only five callback functions that can be registered, a subset of the current twelve.

Here are the prototypes for the 1.0/1.1 tessellator. The 1.0/1.1 tessellator still works in GLU 1.2, but its use is no longer recommended.

void gluBeginPolygon(GLUtriangulatorObj *tessobj); void gluNextContour(GLUtriangulatorObj *tessobj, GLenum type); void gluEndPolygon(GLUtriangulatorObj *tessobj);

The outermost contour must be specified first, and it does not require an initial call to gluNextContour(). For polygons without holes, only one contour is defined, and gluNextContour() is not used. If a polygon has multiple contours (that is, holes or holes within holes), the contours are specified one after the other, each preceded by gluNextContour(). gluTessVertex() is called for each vertex of a contour. For gluNextContour(), type can be GLU_EXTERIOR, GLU_INTERIOR, GLU_CCW, GLU_CW, or GLU_UNKNOWN. These serve only as hints to the tessellation. If you get them right, the tessellation might go faster. If you get them wrong, they're ignored, and the tessellation still works. For polygons with holes, one contour is the exterior contour and the other's interior. The first contour is assumed to be of type GLU_EXTERIOR. Choosing clockwise and counterclockwise orientation is arbitrary in three dimensions; however, there are two different orientations in any plane, and the GLU_CCW and GLU_CW types should be used consistently. Use GLU_UNKNOWN if you don't have a clue.

It is highly recommended that you convert GLU 1.0/1.1 code to the new tessellation interface for GLU 1.2 by following these steps.

- 1. Change references to the major data structure type from GLUtriangulatorObj to GLUtesselator. In GLU 1.2, GLUtriangulatorObj and GLUtesselator are defined to be the same type.
- 2. Convert **gluBeginPolygon**() to two commands: **gluTessBeginPolygon**() and **gluTessBeginContour**(). All contours must be explicitly started, including the first one.
- 3. Convert **gluNextContour**() to both **gluTessEndContour**() and **gluTessBeginContour**(). You have to end the previous contour before starting the next one.
- 4. Convert **gluEndPolygon**() to both **gluTessEndContour**() and **gluTessEndPolygon**(). The final contour must be closed.
- 5. Change references to constants to **gluTessCallback**(). In GLU 1.2, GLU_BEGIN, GLU_VERTEX, GLU_END, GLU_ERROR, and GLU_EDGE_FLAG are defined as synonyms for GLU_TESS_BEGIN, GLU_TESS_VERTEX, GLU_TESS_END, GLU_TESS_ERROR, and GLU_TESS_EDGE_FLAG.

Quadrics: Rendering Spheres, Cylinders, and Disks

The base OpenGL library only provides support for modeling and rendering simple points, lines, and convex filled polygons. Neither 3D objects, nor commonly used 2D objects such as circles, are directly available.

Throughout this book, you've been using GLUT to create some 3D objects. The GLU also provides routines to model and render tessellated, polygonal approximations for a variety of 2D and 3D shapes (spheres, cylinders, disks, and parts of disks), which can be calculated with quadric equations. This includes routines to draw the quadric surfaces in a variety of styles and orientations.

Quadric surfaces are defined by the following general quadratic equation:

a1x2 + a2y2 + a3z2 + a4xy + a5yx + a6xz + a7x + a8y + a9z + a10 = 0

(See David Rogers' *Procedural Elements for Computer Graphics*. New York, NY: McGraw-Hill Book Company, 1985.) Creating and rendering a quadric surface is similar to using the tessellator. To use a quadrics object, follow these steps.

- 1. To create a quadrics object, use gluNewQuadric().
- 2. Specify the rendering attributes for the quadrics object (unless you're satisfied with the default values).
 - 1. Use **gluQuadricOrientation**() to control the winding direction and differentiate the interior from the exterior.
 - 2. Use **gluQuadricDrawStyle**() to choose between rendering the object as points, lines, or filled polygons.
 - 3. For lit quadrics objects, use **gluQuadricNormals**() to specify one normal per vertex or one normal per face. The default is that no normals are generated at all.
 - 4. For textured quadrics objects, use **gluQuadricTexture()** if you want to generate texture coordinates.
- 3. Prepare for problems by registering an error-handling routine with **gluQuadricCallback**(). Then, if an error occurs during rendering, the routine you've specified is invoked.
- 4. Now invoke the rendering routine for the desired type of quadrics object: **gluSphere**(), **gluCylinder**(), **gluDisk**(), or **gluPartialDisk**(). For best performance for static data, encapsulate the quadrics object in a display list.
- 5. When you're completely finished with it, destroy this object with **gluDeleteQuadric()**. If you need to create another quadric, it's best to reuse your quadrics object.

Manage Quadrics Objects

erased.

A quadrics object consists of parameters, attributes, and callbacks that are stored in a data structure of type GLUquadricObj. A quadrics object may generate vertices, normals, texture coordinates, and other data, all of which may be used immediately or stored in a display list for later use. The following routines create, destroy, and report upon errors of a quadrics object.

GLUquadricObj* gluNewQuadric (void);

Creates a new quadrics object and returns a pointer to it. A null pointer is returned if the routine fails.

void gluDeleteQuadric (GLUquadricObj *qobj);

Destroys the quadrics object qobj and frees up any memory used by it.

void **gluQuadricCallback** (GLUquadricObj *qobj, GLenum which, void (*fn)()); Defines a function fn to be called in special circumstances. GLU_ERROR is the only legal value for which, so fn is called when an error occurs. If fn is NULL, any existing callback is For GLU_ERROR, *fn* is called with one parameter, which is the error code. **gluErrorString**() can be used to convert the error code into an ASCII string.

Control Quadrics Attributes

The following routines affect the kinds of data generated by the quadrics routines. Use these routines before you actually specify the primitives.

Example 11-4, quadric.c, on page 435, demonstrates changing the drawing style and the kind of normals generated as well as creating quadrics objects, error handling, and drawing the primitives.

void **gluQuadricDrawStyle** (GLUquadricObj *qobj, GLenum drawStyle); For the quadrics object qobj, drawStyle controls the rendering style. Legal values for drawStyle are GLU_POINT, GLU_LINE, GLU_SILHOUETTE, and GLU_FILL.

GLU_POINT and GLU_LINE specify that primitives should be rendered as a point at every vertex or a line between each pair of connected vertices.

GLU_SILHOUETTE specifies that primitives are rendered as lines, except that edges separating coplanar faces are not drawn. This is most often used for **gluDisk**() and **gluPartialDisk**().

GLU_FILL specifies rendering by filled polygons, where the polygons are drawn in a counterclockwise fashion with respect to their normals. This may be affected by **gluQuadricOrientation()**.

void **gluQuadricOrientation** (GLUquadricObj *qobj, GLenum orientation); For the quadrics object qobj, orientation is either GLU_OUTSIDE (the default) or GLU_INSIDE, which controls the direction in which normals are pointing.

For **gluSphere**() and **gluCylinder**(), the definitions of outside and inside are obvious. For **gluDisk**() and **gluPartialDisk**(), the positive *z* side of the disk is considered to be outside.

void **gluQuadricNormals** (GLUquadricObj *qobj, GLenum normals); For the quadrics object qobj, normals is one of GLU_NONE (the default), GLU_FLAT, or GLU_SMOOTH.

gluQuadricNormals() is used to specify when to generate normal vectors. GLU_NONE means that no normals are generated and is intended for use without lighting. GLU_FLAT generates one normal for each facet, which is often best for lighting with flat shading. GLU_SMOOTH generates one normal for every vertex of the quadric, which is usually best for lighting with smooth shading.

void gluQuadricTexture (GLUquadricObj *qobj,

GLboolean textureCoords);

For the quadrics object qobj, textureCoords is either GL_FALSE (the default) or GL_TRUE. If the value of textureCoords is GL_TRUE, then texture coordinates are generated for the quadrics object. The manner in which the texture coordinates are generated varies, depending upon the type of quadrics object rendered.

Quadrics Primitives

The following routines actually generate the vertices and other data that constitute a quadrics object. In each case, *qobj* refers to a quadrics object created by **gluNewQuadric**().

void gluSphere (GLUquadricObj *qobj, GLdouble radius,

GLint slices, GLint stacks);

Draws a sphere of the given radius, centered around the origin, (0, 0, 0). The sphere is subdivided around the z axis into a number of slices (similar to longitude) and along the z axis into a number of stacks (latitude).

If texture coordinates are also generated by the quadrics facility, the t coordinate ranges from 0.0 at z = -radius to 1.0 at z = radius, with t increasing linearly along longitudinal lines. Meanwhile, s ranges from 0.0 at the +y axis, to 0.25 at the +x axis, to 0.5 at the -y axis, to 0.75 at the -x axis, and back to 1.0 at the +y axis.

void gluCylinder (GLUquadricObj *qobj, GLdouble baseRadius,

GLdouble topRadius, GLdouble height,

GLint slices, GLint stacks);

Draws a cylinder oriented along the z axis, with the base of the cylinder at z = 0 and the top at z = height. Like a sphere, the cylinder is subdivided around the z axis into a number of slices and along the z axis into a number of stacks. baseRadius is the radius of the cylinder at z = 0. topRadius is the radius of the cylinder at z = height. If topRadius is set to zero, then a cone is generated.

If texture coordinates are generated by the quadrics facility, then the t coordinate ranges linearly from 0.0 at z = 0 to 1.0 at z = height. The s texture coordinates are generated the same way as they are for a sphere.

Note: The cylinder is not closed at the top or bottom. The disks at the base and at the top are not drawn.

void gluDisk (GLUquadricObj *qobj, GLdouble innerRadius,

GLdouble outerRadius, GLint slices, GLint rings);

Draws a disk on the z = 0 plane, with a radius of outerRadius and a concentric circular hole with a radius of innerRadius. If innerRadius is 0, then no hole is created. The disk is subdivided around the z axis into a number of slices (like slices of pizza) and also about the z axis into a number of concentric rings.

With respect to orientation, the +z side of the disk is considered to be "outside"; that is, any normals generated point along the +z axis. Otherwise, the normals point along the -z axis. If texture coordinates are generated by the quadrics facility, then the texture coordinates are generated linearly such that where R=outerRadius, the values for s and t at (R, 0, 0) is (1, 0.5), at (0, R, 0) they are (0.5, 1), at (-R, 0, 0) they are (0, 0.5), and at (0, -R, 0) they are (0.5, 0).

void **gluPartialDisk** (GLUquadricObj *qobj, GLdouble innerRadius,

GLdouble outerRadius, GLint slices, GLint rings,

GLdouble startAngle, GLdouble sweepAngle);

Draws a partial disk on the z = 0 plane. A partial disk is similar to a complete disk, in terms of outerRadius, innerRadius, slices, and rings. The difference is that only a portion of a partial disk is drawn, starting from startAngle through startAngle+sweepAngle (where startAngle and sweepAngle are measured in degrees, where 0 degrees is along the +y axis, 90 degrees along the +x axis, 180 along the -y axis, and 270 along the -x axis). A partial disk handles orientation and texture coordinates in the same way as a complete disk.

Note: For all quadrics objects, it's better to use the **Radius*, *height*, and similar arguments to scale them rather than the **glScale***() command so that the unit-length normals that are generated don't

have to be renormalized. Set the *rings* and *stacks* arguments to values other than one to force lighting calculations at a finer granularity, especially if the material specularity is high.

Example 11-4 shows each of the quadrics primitives being drawn, as well as the effects of different drawing styles.

Example 11-4 : Quadrics Objects: quadric.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
GLuint startList;
void errorCallback(GLenum errorCode)
{
   const GLubyte *estring;
   estring = gluErrorString(errorCode);
   fprintf(stderr, "Quadric Error: %s\n", estring);
   exit(0);
}
void init(void)
   GLUquadricObj *qobj;
   GLfloat mat_ambient[] = { 0.5, 0.5, 0.5, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_shininess[] = { 50.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat model_ambient[] = { 0.5, 0.5, 0.5, 1.0 };
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, model_ambient);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
/* Create 4 display lists, each with a different quadric object.
 * Different drawing styles and surface normal specifications
 * are demonstrated.
 * /
   startList = glGenLists(4);
   qobj = gluNewQuadric();
   gluQuadricCallback(qobj, GLU_ERROR, errorCallback);
   gluQuadricDrawStyle(qobj, GLU_FILL); /* smooth shaded */
   gluQuadricNormals(qobj, GLU_SMOOTH);
   glNewList(startList, GL_COMPILE);
      gluSphere(qobj, 0.75, 15, 10);
   glEndList();
   gluQuadricDrawStyle(qobj, GLU_FILL); /* flat shaded */
   gluQuadricNormals(qobj, GLU_FLAT);
   glNewList(startList+1, GL_COMPILE);
```

```
gluCylinder(qobj, 0.5, 0.3, 1.0, 15, 5);
   glEndList();
   gluQuadricDrawStyle(qobj, GLU_LINE); /* wireframe */
   gluQuadricNormals(qobj, GLU_NONE);
   glNewList(startList+2, GL_COMPILE);
      gluDisk(qobj, 0.25, 1.0, 20, 4);
   glEndList();
   gluQuadricDrawStyle(qobj, GLU_SILHOUETTE);
   gluQuadricNormals(qobj, GLU_NONE);
   glNewList(startList+3, GL_COMPILE);
      gluPartialDisk(qobj, 0.0, 1.0, 20, 4, 0.0, 225.0);
  glEndList();
}
void display(void)
{
  glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glEnable(GL_LIGHTING);
  glShadeModel (GL SMOOTH);
  glTranslatef(-1.0, -1.0, 0.0);
  glCallList(startList);
  glShadeModel (GL_FLAT);
  glTranslatef(0.0, 2.0, 0.0);
  glPushMatrix();
  glRotatef(300.0, 1.0, 0.0, 0.0);
  glCallList(startList+1);
  glPopMatrix();
  glDisable(GL LIGHTING);
  glColor3f(0.0, 1.0, 1.0);
  glTranslatef(2.0, -2.0, 0.0);
  glCallList(startList+2);
  glColor3f(1.0, 1.0, 0.0);
   glTranslatef(0.0, 2.0, 0.0);
  glCallList(startList+3);
   glPopMatrix();
   glFlush();
ł
void reshape (int w, int h)
ł
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
   if (w <= h)
      glOrtho(-2.5, 2.5, -2.5*(GLfloat)h/(GLfloat)w,
         2.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho(-2.5*(GLfloat)w/(GLfloat)h,
         2.5*(GLfloat)w/(GLfloat)h, -2.5, 2.5, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
void keyboard(unsigned char key, int x, int y)
ł
   switch (key) {
      case 27:
```

```
exit(0);
         break;
   }
}
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(100, 100);
  glutCreateWindow(argv[0]);
  init();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

+ >

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 12 Evaluators and NURBS

Chapter Objectives

Advanced

After reading this chapter, you'll be able to do the following:

- Use OpenGL evaluator commands to draw basic curves and surfaces
- Use the GLU's higher-level NURBS facility to draw more complex curves and surfaces

Note that this chapter presumes a number of prerequisites; they're listed in "Prerequisites."

At the lowest level, graphics hardware draws points, line segments, and polygons, which are usually triangles and quadrilaterals. Smooth curves and surfaces are drawn by approximating them with large numbers of small line segments or polygons. However, many useful curves and surfaces can be described mathematically by a small number of parameters such as a few *control points*. Saving the 16 control points for a surface requires much less storage than saving 1000 triangles together with the normal vector information at each vertex. In addition, the 1000 triangles only approximate the true surface, but the control points accurately describe the real surface.

Evaluators provide a way to specify points on a curve or surface (or part of one) using only the control points. The curve or surface can then be rendered at any precision. In addition, normal vectors can be calculated for surfaces automatically. You can use the points generated by an evaluator in many ways - to draw dots where the surface would be, to draw a wireframe version of the surface, or to draw a fully lighted, shaded, and even textured version.

You can use evaluators to describe any polynomial or rational polynomial splines or surfaces of any degree. These include almost all splines and spline surfaces in use today, including B-splines, NURBS (Non-Uniform Rational B-Spline) surfaces, Bézier curves and surfaces, and Hermite splines. Since evaluators provide only a low-level description of the points on a curve or surface, they're typically used underneath utility libraries that provide a higher-level interface to the programmer. The GLU's NURBS facility is such a higher-level interface - the NURBS routines encapsulate lots of complicated code. Much of the final rendering is done with evaluators, but for some conditions (trimming curves, for example) the NURBS routines use planar polygons for rendering.

This chapter contains the following major sections.

• "Prerequisites" discusses what knowledge is assumed for this chapter. It also gives several references where you can obtain this information.

- "Evaluators" explains how evaluators work and how to control them using the appropriate OpenGL commands.
- "The GLU NURBS Interface" describes the GLU routines for creating NURBS surfaces.

Prerequisites

Evaluators make splines and surfaces that are based on a Bézier (or Bernstein) basis. The defining formulas for the functions in this basis are given in this chapter, but the discussion doesn't include derivations or even lists of all their interesting mathematical properties. If you want to use evaluators to draw curves and surfaces using other bases, you must know how to convert your basis to a Bézier basis. In addition, when you render a Bézier surface or part of it using evaluators, you need to determine the granularity of your subdivision. Your decision needs to take into account the trade-off between high-quality (highly subdivided) images and high speed. Determining an appropriate subdivision strategy can be quite complicated - too complicated to be discussed here.

Similarly, a complete discussion of NURBS is beyond the scope of this book. The GLU NURBS interface is documented here, and programming examples are provided for readers who already understand the subject. In what follows, you already should know about NURBS control points, knot sequences, and trimming curves.

If you lack some of these prerequisites, the following references will help.

- Farin, Gerald E., *Curves and Surfaces for Computer-Aided Geometric Design, Fourth Edition.* San Diego, CA: Academic Press, 1996.
- Farin, Gerald E., *NURB Curves and Surfaces: from Projective Geometry to Practical Use.* Wellesley, MA: A. K. Peters Ltd., 1995.
- Farin, Gerald E., editor, *NURBS for Curve and Surface Design*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.
- Hoschek, Josef and Dieter Lasser, *Fundamentals of Computer Aided Geometric Design*. Wellesley, MA: A. K. Peters Ltd., 1993.
- Piegl, Les and Wayne Tiller, *The NURBS Book*. New York, NY: Springer-Verlag, 1995.

Note: Some terms used in this chapter might have slightly different meanings in other books on spline curves and surfaces, since there isn't total agreement among the practitioners of this art. Generally, the OpenGL meanings are a bit more restrictive. For example, OpenGL evaluators always use Bézier bases; in other contexts, evaluators might refer to the same concept, but with an arbitrary basis.

Evaluators

A Bézier curve is a vector-valued function of one variable

 $\mathbf{C}(u) = [\mathbf{X}(u) \ \mathbf{Y}(u) \ \mathbf{Z}(u)]$

where u varies in some domain (say [0,1]). A Bézier surface patch is a vector-valued function of two variables

 $\mathbf{S}(u,v) = [\mathbf{X}(u,v) \ \mathbf{Y}(u,v) \ \mathbf{Z}(u,v)]$

where u and v can both vary in some domain. The range isn't necessarily three-dimensional as shown here. You might want two-dimensional output for curves on a plane or texture coordinates, or you might want four-dimensional output to specify RGBA information. Even one-dimensional output may make sense for gray levels.

For each u (or u and v, in the case of a surface), the formula for C() (or S()) calculates a point on the curve (or surface). To use an evaluator, first define the function C() or S(), enable it, and then use the **glEvalCoord1**() or **glEvalCoord2**() command instead of **glVertex***(). This way, the curve or surface vertices can be used like any other vertices - to form points or lines, for example. In addition, other commands automatically generate series of vertices that produce a regular mesh uniformly spaced in u (or in u and v). One- and two-dimensional evaluators are similar, but the description is somewhat simpler in one dimension, so that case is discussed first.

One-Dimensional Evaluators

This section presents an example of using one-dimensional evaluators to draw a curve. It then describes the commands and equations that control evaluators.

One-Dimensional Example: A Simple Bézier Curve

The program shown in Example 12-1 draws a cubic Bézier curve using four control points, as shown in Figure 12-1.

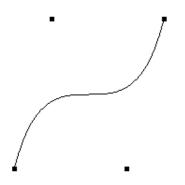


Figure 12-1 : Bézier Curve

Example 12-1 : Bézier Curve with Four Control Points: bezcurve.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>
GLfloat ctrlpoints[4][3] = {
```

```
\{-4.0, -4.0, 0.0\}, \{-2.0, 4.0, 0.0\},
        \{2.0, -4.0, 0.0\}, \{4.0, 4.0, 0.0\}\};
void init(void)
ł
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}
void display(void)
{
   int i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord1f((GLfloat) i/30.0);
   glEnd();
   /* The following code displays the control points as dots. */
   glPointSize(5.0);
   glColor3f(1.0, 1.0, 0.0);
   glBegin(GL POINTS);
      for (i = 0; i < 4; i++)
         glVertex3fv(&ctrlpoints[i][0]);
   glEnd();
   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL PROJECTION);
   glLoadIdentity();
   if (w \le h)
      glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
               5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
   else
      glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
               5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
int main(int argc, char** argv)
ł
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

A cubic Bézier curve is described by four control points, which appear in this example in the *ctrlpoints[][]* array. This array is one of the arguments to **glMap1f**(). All the arguments for this command are as follows:

GL_MAP1_VERTEX_3

Three-dimensional control points are provided and three-dimensional vertices are produced

0.0

Low value of parameter *u*

1.0

High value of parameter *u*

3

The number of floating-point values to advance in the data between one control point and the next

4

The order of the spline, which is the degree+1: in this case, the degree is 3 (since this is a cubic curve)

&ctrlpoints[0][0]

Pointer to the first control point's data

Note that the second and third arguments control the parameterization of the curve - as the variable u ranges from 0.0 to 1.0, the curve goes from one end to the other. The call to **glEnable**() enables the one-dimensional evaluator for three-dimensional vertices.

The curve is drawn in the routine **display**() between the **glBegin**() and **glEnd**() calls. Since the evaluator is enabled, the command **glEvalCoord1f**() is just like issuing a **glVertex**() command with the coordinates of a vertex on the curve corresponding to the input parameter *u*.

Defining and Evaluating a One-Dimensional Evaluator

The Bernstein polynomial of degree n (or order n+1) is given by

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

If Pi represents a set of control points (one-, two-, three-, or even four- dimensional), then the equation

$$C(u) = \prod_{i=0}^{n} B_{i}^{n}(u) P_{i}$$

represents a Bézier curve as u varies from 0.0 to 1.0. To represent the same curve but allowing u to vary between u1 and u2 instead of 0.0 and 1.0, evaluate

$$C\left(\frac{u-u_1}{u_2-u_1}\right)$$

The command **glMap1**() defines a one-dimensional evaluator that uses these equations.

void glMap1{fd}(GLenum target, TYPEu1, TYPEu2, GLint stride,

GLint order, const TYPE*points);

Defines a one-dimensional evaluator. The target parameter specifies what the control points represent, as shown in Table 12-1, and therefore how many values need to be supplied in points. The points can represent vertices, RGBA color data, normal vectors, or texture coordinates. For

example, with GL_MAP1_COLOR_4, the evaluator generates color data along a curve in four-dimensional (RGBA) color space. You also use the parameter values listed in Table 12-1 to enable each defined evaluator before you invoke it. Pass the appropriate value to **glEnable()** or **glDisable()** to enable or disable the evaluator.

The second two parameters for **glMap1***(), u1 and u2, indicate the range for the variable u. The variable stride is the number of single- or double-precision values (as appropriate) in each block of storage. Thus, it's an offset value between the beginning of one control point and the beginning of the next.

The order is the degree plus one, and it should agree with the number of control points. The points parameter points to the first coordinate of the first control point. Using the example data structure for **glMap1***(), use the following for points:

(GLfloat *)(&ctlpoints[0].x)

Parameter	Meaning
GL_MAP1_VERTEX_3	<i>x</i> , <i>y</i> , <i>z</i> vertex coordinates
GL_MAP1_VERTEX_4	<i>x, y, z, w</i> vertex coordinates
GL_MAP1_INDEX	color index
GL_MAP1_COLOR_4	R, G, B, A
GL_MAP1_NORMAL	normal coordinates
GL_MAP1_TEXTURE_COORD_1	s texture coordinates
GL_MAP1_TEXTURE_COORD_2	<i>s</i> , <i>t</i> texture coordinates
GL_MAP1_TEXTURE_COORD_3	<i>s, t, r</i> texture coordinates
GL_MAP1_TEXTURE_COORD_4	s, t, r, q texture coordinates

Table 12-1 : Types of Control Points for glMap1*()

More than one evaluator can be evaluated at a time. If you have both a GL_MAP1_VERTEX_3 and a GL_MAP1_COLOR_4 evaluator defined and enabled, for example, then calls to **glEvalCoord1(**) generate both a position and a color. Only one of the vertex evaluators can be enabled at a time, although you might have defined both of them. Similarly, only one of the texture evaluators can be active. Other than that, however, evaluators can be used to generate any combination of vertex, normal, color, and texture-coordinate data. If more than one evaluator of the same type is defined and enabled, the one of highest dimension is used.

Use **glEvalCoord1***() to evaluate a defined and enabled one-dimensional map.

void glEvalCoord1{fd}(TYPE u);

```
void glEvalCoord1{fd}v(TYPE *u);
```

Causes evaluation of the enabled one-dimensional maps. The argument u is the value (or a pointer to the value, in the vector version of the command) of the domain coordinate.

For evaluated vertices, values for color, color index, normal vectors, and texture coordinates are generated by evaluation. Calls to **glEvalCoord***() do not use the current values for color, color index, normal vectors, and texture coordinates. **glEvalCoord***() also leaves those values unchanged.

Defining Evenly Spaced Coordinate Values in One Dimension

You can use **glEvalCoord1**() with any values for *u*, but by far the most common use is with evenly spaced values, as shown previously in Example 12-1. To obtain evenly spaced values, define a one-dimensional grid using **glMapGrid1***() and then apply it using **glEvalMesh1**().

```
void glMapGrid1{fd}(GLint n, TYPEu1, TYPEu2);
Defines a grid that goes from u1 to u2 in n steps, which are evenly spaced.
void glEvalMesh1(GLenum mode, GLint p1, GLint p2);
Applies the currently defined map grid to all enabled evaluators. The mode can be either
```

Applies the currently defined map grid to all enabled evaluators. The mode can be either GL_POINT or GL_LINE , depending on whether you want to draw points or a connected line along the curve. The call has exactly the same effect as issuing a **glEvalCoord1**() for each of the steps between and including p1 and p2, where $0 \le p1$, $p2 \le n$. Programmatically, it's equivalent to the following:

```
glBegin(GL_POINTS);  /* OR glBegin(GL_LINE_STRIP); */
for (i = p1; i <= p2; i++)
    glEvalCoordl(u1 + i*(u2-u1)/n);
glEnd();</pre>
```

except that if i = 0 or i = n, then **glEvalCoord1**() is called with exactly u1 or u2 as its parameter.

Two-Dimensional Evaluators

In two dimensions, everything is similar to the one-dimensional case, except that all the commands must take two parameters, u and v, into account. Points, colors, normals, or texture coordinates must be supplied over a surface instead of a curve. Mathematically, the definition of a Bézier surface patch is given by

$$S(u, v) = B_i^n(u) B_j^m(v) P_{ij}$$
$$i=0 \ j=0$$

where P_{ij} are a set of m*n control points, and the B_i are the same Bernstein polynomials for one dimension. As before, the P_{ij} can represent vertices, normals, colors, or texture coordinates.

The procedure to use two-dimensional evaluators is similar to the procedure for one dimension.

- 1. Define the evaluator(s) with **glMap2***().
- 2. Enable them by passing the appropriate value to **glEnable**().
- 3. Invoke them either by calling **glEvalCoord2**() between a **glBegin**() and **glEnd**() pair or by specifying and then applying a mesh with **glMapGrid2**() and **glEvalMesh2**().

Defining and Evaluating a Two-Dimensional Evaluator

Use glMap2*() and glEvalCoord2*() to define and then invoke a two-dimensional evaluator.

void **glMap2**{fd}(GLenum target, TYPEu1, TYPEu2, GLint ustride, GLint uorder, TYPEv1, TYPEv2, GLint vstride, GLint vorder, TYPE points);

The target parameter can have any of the values in Table 12-1, except that the string MAP1 is replaced with MAP2. As before, these values are also used with **glEnable**() to enable the corresponding evaluator. Minimum and maximum values for both u and v are provided as u1, u2, v1, and v2. The parameters ustride and vstride indicate the number of single- or double-precision values (as appropriate) between independent settings for these values, allowing users to select a subrectangle of control points out of a much larger array. For example, if the data appears in the form

GLfloat ctlpoints[100][100][3];

and you want to use the 4x4 subset beginning at ctlpoints[20][30], choose ustride to be 100*3 and vstride to be 3. The starting point, points, should be set to &ctlpoints[20][30][0]. Finally, the order parameters, uorder and vorder, can be different, allowing patches that are cubic in one direction and quadratic in the other, for example.

void glEvalCoord2{fd}(TYPE u, TYPE v);

void glEvalCoord2{fd}v(TYPE *values);

Causes evaluation of the enabled two-dimensional maps. The arguments u and v are the values (or a pointer to the values u and v, in the vector version of the command) for the domain coordinates. If either of the vertex evaluators is enabled (GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4), then the normal to the surface is computed analytically. This normal is associated with the generated vertex if automatic normal generation has been enabled by passing GL_AUTO_NORMAL to glEnable(). If it's disabled, the corresponding enabled normal map is used to produce a normal. If no such map exists, the current normal is used.

Two-Dimensional Example: A Bézier Surface

Example 12-2 draws a wireframe Bézier surface using evaluators, as shown in Figure 12-2. In this example, the surface is drawn with nine curved lines in each direction. Each curve is drawn as 30 segments. To get the whole program, add the **reshape()** and **main()** routines from Example 12-1.

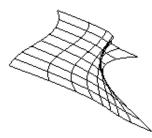


Figure 12-2 : Bézier Surface

Example 12-2 : Bézier Surface: bezsurf.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>
GLfloat ctrlpoints[4][4][3] = {
            \{\{-1.5, -1.5, 4.0\}, \{-0.5, -1.5, 2.0\},\
                   0.5, -1.5, -1.0, \{1.5, -1.5, 2.0\},
            \{\{-1.5, -0.5, 1.0\}, \{-0.5, -0.5, 3.0\},\
                  \{0.5, -0.5, 0.0\}, \{1.5, -0.5, -1.0\}\},\
            \{\{-1.5, 0.5, 4.0\}, \{-0.5, 0.5, 0.0\},\
            \{0.5, 0.5, 3.0\}, \{1.5, 0.5, 4.0\}\}, \{\{-1.5, 1.5, -2.0\}, \{-0.5, 1.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.0\}, \{-0.5, -2.
                 \{0.5, 1.5, 0.0\}, \{1.5, 1.5, -1.0\}\}
};
void display(void)
ł
           int i, j;
           glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
           glColor3f(1.0, 1.0, 1.0);
           glPushMatrix ();
           glRotatef(85.0, 1.0, 1.0, 1.0);
           for (j = 0; j <= 8; j++) {
                       glBegin(GL_LINE_STRIP);
                       for (i = 0; i <= 30; i++)
                                  glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
                       glEnd();
                       glBegin(GL_LINE_STRIP);
                       for (i = 0; i <= 30; i++)
                                  glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
                       glEnd();
            }
           glPopMatrix ();
           glFlush();
}
void init(void)
ł
           glClearColor (0.0, 0.0, 0.0, 0.0);
```

```
glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
                0, 1, 12, 4, &ctrlpoints[0][0][0]);
glEnable(GL_MAP2_VERTEX_3);
glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
glEnable(GL_DEPTH_TEST);
glShadeModel(GL_FLAT);
```

}

Defining Evenly Spaced Coordinate Values in Two Dimensions

In two dimensions, the **glMapGrid2***() and **glEvalMesh2**() commands are similar to the one-dimensional versions, except that both *u* and *v* information must be included.

```
void glMapGrid2{fd}(GLint nu, TYPEu1, TYPEu2,
GLint nv, TYPEv1, TYPEv2);
void glEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);
Defines a two-dimensional map grid that goes from u1 to u2 in nu evenly spaced steps, from
v1 to v2 in nv steps (glMapGrid2*()), and then applies this grid to all enabled evaluators
(glEvalMesh2()). The only significant difference from the one-dimensional versions of these
two commands is that in glEvalMesh2() the mode parameter can be GL_FILL as well as
GL_POINT or GL_LINE. GL_FILL generates filled polygons using the quad-mesh primitive.
Stated precisely, glEvalMesh2() is nearly equivalent to one of the following three code
fragments. (It's nearly equivalent because when i is equal to nu or j to nv, the parameter is
exactly equal to u2 or v2, not to u1+nu*(u2-u1)/nu, which might be slightly different due to
round-off error.)
```

```
glBegin(GL_POINTS);
                                      /* mode == GL POINT */
for (i = nu1; i <= nu2; i++)</pre>
    for (j = nv1; j <= nv2; j++)</pre>
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();
or
for (i = nul; i <= nu2; i++) {</pre>
                                      /* mode == GL_LINE */
    glBegin(GL_LINES);
        for (j = nv1; j <= nv2; j++)</pre>
             glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
for (j = nv1; j <= nv2; j++) {
    glBegin(GL_LINES);
    for (i = nu1; i <= nu2; i++)</pre>
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
}
or
for (i = nu1; i < nu2; i++) {</pre>
                                     /* mode == GL FILL */
    glBegin(GL_QUAD_STRIP);
    for (j = nv1; j <= nv2; j++) {</pre>
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
        glEvalCoord2(u1 + (i+1)*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
}
```

Example 12-3 shows the differences necessary to draw the same Bézier surface as Example 12-2, but using **glMapGrid2()** and **glEvalMesh2()** to subdivide the square domain into a uniform 8x8

grid. This program also adds lighting and shading, as shown in Figure 12-3.



Figure 12-3 : Lit, Shaded Bézier Surface Drawn with a Mesh

Example 12-3 : Lit, Shaded Bézier Surface Using a Mesh: bezmesh.c

```
void initlights(void)
{
   GLfloat ambient[] = \{0.2, 0.2, 0.2, 1.0\};
   GLfloat position[] = \{0.0, 0.0, 2.0, 1.0\};
  GLfloat mat_diffuse[] = {0.6, 0.6, 0.6, 1.0};
   GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
   GLfloat mat_shininess[] = {50.0};
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT0, GL_POSITION, position);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}
void display(void)
{
   glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
  glPushMatrix();
  glRotatef(85.0, 1.0, 1.0, 1.0);
   glEvalMesh2(GL_FILL, 0, 20, 0, 20);
   glPopMatrix();
   glFlush();
}
void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
  glEnable(GL DEPTH TEST);
   glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
           0, 1, 12, 4, &ctrlpoints[0][0][0]);
   glEnable(GL_MAP2_VERTEX_3);
   glEnable(GL_AUTO_NORMAL);
   glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
   initlights();
}
```

Using Evaluators for Textures

Example 12-4 enables two evaluators at the same time: The first generates three-dimensional points on the same Bézier surface as Example 12-3, and the second generates texture coordinates. In this case, the texture coordinates are the same as the u and v coordinates of the surface, but a special flat Bézier patch must be created to do this.

The flat patch is defined over a square with corners at (0, 0), (0, 1), (1, 0), and (1, 1); it generates (0, 0) at corner (0, 0), (0, 1) at corner (0, 1), and so on. Since it's of order two (linear degree plus one), evaluating this texture at the point (u, v) generates texture coordinates (s, t). It's enabled at the same time as the vertex evaluator, so both take effect when the surface is drawn. (See "Plate 19" in Appendix I.) If you want the texture to repeat three times in each direction, change every 1.0 in the array *texpts*[*]*[*]*[*]* to 3.0. Since the texture wraps in this example, the surface is rendered with nine copies of the texture map.

Example 12-4 : Using Evaluators for Textures: texturesurf.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>
#include <math.h>
GLfloat ctrlpoints[4][4][3] = {
    \{\{-1.5, -1.5, 4.0\}, \{-0.5, -1.5, 2.0\}, \{0.5, -1.5, -1.0\}, \{1.5, -1.5, 2.0\}\},\
    \{\{-1.5, -0.5, 1.0\}, \{-0.5, -0.5, 3.0\}, \{0.5, -0.5, 0.0\}, \{1.5, -0.5, -1.0\}\}, \{\{-1.5, 0.5, 4.0\}, \{-0.5, 0.5, 0.0\}, 0.0\}, \{-1.5, 0.5, 0.5, 0.0\}, 0.0\}
     \{0.5, 0.5, 3.0\}, \{1.5, 0.5, 4.0\}\}, \\ \{\{-1.5, 1.5, -2.0\}, \{-0.5, 1.5, -2.0\}, \\ \{0.5, 1.5, 0.0\}, \{1.5, 1.5, -1.0\}\} 
};
GLfloat texpts[2][2][2] = \{\{\{0.0, 0.0\}, \{0.0, 1.0\}\}, \}
                                \{\{1.0, 0.0\}, \{1.0, 1.0\}\}\};
void display(void)
{
   glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
   glColor3f(1.0, 1.0, 1.0);
   glEvalMesh2(GL_FILL, 0, 20, 0, 20);
   glFlush();
#define imageWidth 64
#define imageHeight 64
GLubyte image[3*imageWidth*imageHeight];
void makeImage(void)
    int i, j;
    float ti, tj;
    for (i = 0; i < imageWidth; i++) {</pre>
        ti = 2.0*3.14159265*i/imageWidth;
        for (j = 0; j < imageHeight; j++) {
            tj = 2.0*3.14159265*j/imageHeight;
            image[3*(imageHeight*i+j)] =
                  (GLubyte) 127*(1.0+sin(ti));
            image[3*(imageHeight*i+j)+1] =
                  (GLubyte) 127*(1.0+cos(2*tj));
            image[3*(imageHeight*i+j)+2] =
                  (GLubyte) 127*(1.0+cos(ti+tj));
        }
```

```
}
}
void init(void)
   glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
           0, 1, 12, 4, &ctrlpoints[0][0][0]);
   glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2,
           0, 1, 4, 2, &texpts[0][0][0]);
   glEnable(GL_MAP2_TEXTURE_COORD_2);
   glEnable(GL_MAP2_VERTEX_3);
   glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
   makeImage();
   glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, 3, imageWidth, imageHeight, 0,
                GL_RGB, GL_UNSIGNED_BYTE, image);
   glEnable(GL TEXTURE 2D);
  glEnable(GL_DEPTH_TEST);
   glShadeModel (GL_FLAT);
}
void reshape(int w, int h)
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL PROJECTION);
   glLoadIdentity();
   if (w \le h)
      qlOrtho(-4.0, 4.0, -4.0*(GLfloat)h/(GLfloat)w,
              4.0*(GLfloat)h/(GLfloat)w, -4.0, 4.0);
   else
      glOrtho(-4.0*(GLfloat)w/(GLfloat)h,
              4.0*(GLfloat)w/(GLfloat)h, -4.0, 4.0, -4.0, 4.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glRotatef(85.0, 1.0, 1.0, 1.0);
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
   glutInitDisplayMode (GLUT SINGLE | GLUT RGB | GLUT DEPTH);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

The GLU NURBS Interface

Although evaluators are the only OpenGL primitive available to draw curves and surfaces directly, and even though they can be implemented very efficiently in hardware, they're often accessed by

applications through higher-level libraries. The GLU provides a NURBS (Non-Uniform Rational B-Spline) interface built on top of the OpenGL evaluator commands.

A Simple NURBS Example

If you understand NURBS, writing OpenGL code to manipulate NURBS curves and surfaces is relatively easy, even with lighting and texture mapping. Follow these steps to draw NURBS curves or untrimmed NURBS surfaces. (See "Trim a NURBS Surface" for information about trimmed surfaces.)

- 1. If you intend to use lighting with a NURBS surface, call **glEnable**() with GL_AUTO_NORMAL to automatically generate surface normals. (Or you can calculate your own.)
- 2. Use **gluNewNurbsRenderer**() to create a pointer to a NURBS object, which is referred to when creating your NURBS curve or surface.
- 3. If desired, call **gluNurbsProperty**() to choose rendering values, such as the maximum size of lines or polygons that are used to render your NURBS object.
- 4. Call **gluNurbsCallback**() if you want to be notified when an error is encountered. (Error checking may slightly degrade performance but is still highly recommended.)
- 5. Start your curve or surface by calling gluBeginCurve() or gluBeginSurface().
- 6. Generate and render your curve or surface. Call **gluNurbsCurve()** or **gluNurbsSurface()** at least once with the control points (rational or nonrational), knot sequence, and order of the polynomial basis function for your NURBS object. You might call these functions additional times to specify surface normals and/or texture coordinates.
- 7. Call **gluEndCurve()** or **gluEndSurface()** to complete the curve or surface.

Example 12-5 renders a NURBS surface in the shape of a symmetrical hill with control points ranging from -3.0 to 3.0. The basis function is a cubic B-spline, but the knot sequence is nonuniform, with a multiplicity of 4 at each endpoint, causing the basis function to behave like a Bézier curve in each direction. The surface is lighted, with a dark gray diffuse reflection and white specular highlights. Figure 12-4 shows the surface as a lit wireframe.

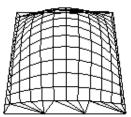


Figure 12-4 : NURBS Surface

Example 12-5 : NURBS Surface: surface.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
GLfloat ctlpoints[4][4][3];
int showPoints = 0;
GLUnurbsObj *theNurb;
void init_surface(void)
{
   int u, v;
  for (u = 0; u < 4; u++) {
      for (v = 0; v < 4; v++) {
         ctlpoints[u][v][0] = 2.0*((GLfloat)u - 1.5);
         ctlpoints[u][v][1] = 2.0*((GLfloat)v - 1.5);
         if (u == 1 || u == 2) \& (v == 1 || v == 2))
            ctlpoints[u][v][2] = 3.0;
         else
            ctlpoints[u][v][2] = -3.0;
      }
   }
}
void nurbsError(GLenum errorCode)
{
   const GLubyte *estring;
   estring = gluErrorString(errorCode);
   fprintf (stderr, "Nurbs Error: %s\n", estring);
   exit (0);
}
void init(void)
{
  GLfloat mat_diffuse[] = { 0.7, 0.7, 0.7, 1.0 };
  GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
  GLfloat mat_shininess[] = { 100.0 };
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_AUTO_NORMAL);
  glEnable(GL_NORMALIZE);
   init_surface();
   theNurb = gluNewNurbsRenderer();
   gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
   gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
  gluNurbsCallback(theNurb, GLU_ERROR,
                    (GLvoid (*)()) nurbsError);
}
void display(void)
{
  GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
```

```
int i, j;
   glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT);
   glPushMatrix();
   glRotatef(330.0, 1.,0.,0.);
  glScalef (0.5, 0.5, 0.5);
   gluBeginSurface(theNurb);
  gluNurbsSurface(theNurb,
                   8, knots, 8, knots,
                   4 * 3, 3, &ctlpoints[0][0][0],
                   4, 4, GL_MAP2_VERTEX_3);
   gluEndSurface(theNurb);
   if (showPoints) {
     glPointSize(5.0);
     glDisable(GL_LIGHTING);
     glColor3f(1.0, 1.0, 0.0);
     glBegin(GL_POINTS);
      for (i = 0; i < 4; i++) {
         for (j = 0; j < 4; j++) {
            glVertex3f(ctlpoints[i][j][0],
                       ctlpoints[i][j][1], ctlpoints[i][j][2]);
         }
      }
      glEnd();
      glEnable(GL_LIGHTING);
   }
  glPopMatrix();
  glFlush();
}
void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
  gluPerspective (45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef (0.0, 0.0, -5.0);
}
void keyboard(unsigned char key, int x, int y)
ł
   switch (key) {
     case `c':
      case 'C':
         showPoints = !showPoints;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
```

```
glutCreateWindow(argv[0]);
init();
glutReshapeFunc(reshape);
glutDisplayFunc(display);
glutKeyboardFunc (keyboard);
glutMainLoop();
return 0;
```

Manage a NURBS Object

}

As shown in Example 12-5, **gluNewNurbsRenderer()** returns a new NURBS object, whose type is a pointer to a GLUnurbsObj structure. You must make this object before using any other NURBS routine. When you're done with a NURBS object, you may use **gluDeleteNurbsRenderer()** to free up the memory that was used.

GLUnurbsObj* gluNewNurbsRenderer (void);

Creates a new NURBS object, nobj. Returns a pointer to the new object, or zero, if OpenGL cannot allocate memory for a new NURBS object. void gluDeleteNurbsRenderer (GLUnurbsObj *nobj); Destroys the NURBS object nobj.

Control NURBS Rendering Properties

A set of properties associated with a NURBS object affects the way the object is rendered. These properties include how the surface is rasterized (for example, filled or wireframe) and the precision of tessellation.

void gluNurbsProperty(GLUnurbsObj *nobj, GLenum property,

GLfloat value);

Controls attributes of a NURBS object, nobj. The property argument specifies the property and can be GLU_DISPLAY_MODE, GLU_CULLING, GLU_SAMPLING_METHOD, GLU_SAMPLING_TOLERANCE, GLU_PARAMETRIC_TOLERANCE, GLU_U_STEP, GLU_V_STEP, or GLU_AUTO_LOAD_MATRIX. The value argument indicates what the property should be.

The default value for GLU_DISPLAY_MODE is GLU_FILL, which causes the surface to be rendered as polygons. If GLU_OUTLINE_POLYGON is used for the display-mode property, only the outlines of polygons created by tessellation are rendered. GLU_OUTLINE_PATCH renders the outlines of patches and trimming curves. (See "Create a NURBS Curve or Surface".)

GLU_CULLING can speed up performance by not performing tessellation if the NURBS object falls completely outside the viewing volume; set this property to GL_TRUE to enable culling (the default is GL_FALSE).

Since a NURBS object is rendered as primitives, it's sampled at different values of its parameter(s) (u and v) and broken down into small line segments or polygons for rendering. If property is GLU_SAMPLING_METHOD, then value is set to one of

GLU_PATH_LENGTH (which is the default), GLU_PARAMETRIC_ERROR, or GLU_DOMAIN_DISTANCE, which specifies how a NURBS curve or surface should be tessellated. When value is set to GLU_PATH_LENGTH, the surface is rendered so that the maximum length, in pixels, of the edges of tessellated polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE. When set to GLU_PARAMETRIC_ERROR, then the value specified by GLU_PARAMETRIC_TOLERANCE is the maximum distance, in pixels, between tessellated polygons and the surfaces they approximate. When set to GLU_DOMAIN_DISTANCE, the application specifies, in parametric coordinates, how many sample points per unit length are taken in the u and v dimensions, using the values for GLU_U_STEP and GLU_V_STEP.

If property is GLU_SAMPLING_TOLERANCE and the sampling method is GLU_PATH_LENGTH, value controls the maximum length, in pixels, to use for tessellated polygons. The default value of 50.0 makes the largest sampled line segment or polygon edge 50.0 pixels long. If property is GLU_PARAMETRIC_TOLERANCE and the sampling method is GLU_PARAMETRIC_ERROR, value controls the maximum distance, in pixels, between the tessellated polygons and the surfaces they approximate. The default value for

GLU_PARAMETRIC_TOLERANCE is 0.5, which makes the tessellated polygons within one-half pixel of the approximated surface. If the sampling method is

GLU_DOMAIN_DISTANCE and property is either GLU_U_STEP or GLU_V_STEP, then value is the number of sample points per unit length taken along the u or v dimension, respectively, in parametric coordinates. The default for both GLU_U_STEP and GLU_V_STEP is 100.

The GLU_AUTO_LOAD_MATRIX property determines whether the projection matrix, modelview matrix, and viewport are downloaded from the OpenGL server (GL_TRUE, the default), or whether the application must supply these matrices with gluLoadSamplingMatrices() (GL_FALSE).

void **gluLoadSamplingMatrices** (GLUnurbsObj *nobj, const GLfloat modelMatrix[16], const GLfloat projMatrix[16], const GLint viewport[4]);

If the GLU_AUTO_LOAD_MATRIX is turned off, the modelview and projection matrices and the viewport specified in **gluLoadSamplingMatrices**() are used to compute sampling and culling matrices for each NURBS curve or surface.

If you need to query the current value for a NURBS property, you may use **gluGetNurbsProperty()**.

void gluGetNurbsProperty (GLUnurbsObj *nobj, GLenum property,

GLfloat *value);

Given the property to be queried for the NURBS object nobj, return its current value.

Handle NURBS Errors

Since there are 37 different errors specific to NURBS functions, it's a good idea to register an error callback to let you know if you've stumbled into one of them. In Example 12-5, the callback function was registered with

gluNurbsCallback(theNurb, GLU_ERROR, (GLvoid (*)()) nurbsError);

void gluNurbsCallback (GLUnurbsObj *nobj, GLenum which,

void (*fn)(GLenum errorCode));

which is the type of callback; it must be GLU_ERROR. When a NURBS function detects an error condition, fn is invoked with the error code as its only argument. errorCode is one of 37 error conditions, named GLU_NURBS_ERROR1 through GLU_NURBS_ERROR37. Use gluErrorString() to describe the meaning of those error codes.

In Example 12-5, the **nurbsError**() routine was registered as the error callback function:

```
void nurbsError(GLenum errorCode)
{
    const GLubyte *estring;
```

```
estring = gluErrorString(errorCode);
fprintf (stderr, "Nurbs Error: %s\n", estring);
exit (0);
}
```

Create a NURBS Curve or Surface

To render a NURBS surface, **gluNurbsSurface**() is bracketed by **gluBeginSurface**() and **gluEndSurface**(). The bracketing routines save and restore the evaluator state.

void gluBeginSurface (GLUnurbsObj *nobj);

void gluEndSurface (GLUnurbsObj *nobj);

After gluBeginSurface(), one or more calls to gluNurbsSurface() defines the attributes of the surface. Exactly one of these calls must have a surface type of GL_MAP2_VERTEX_3 or GL_MAP2_VERTEX_4 to generate vertices. Use gluEndSurface() to end the definition of a surface. Trimming of NURBS surfaces is also supported between gluBeginSurface() and gluEndSurface(). (See "Trim a NURBS Surface".)

void gluNurbsSurface (GLUnurbsObj *nobj, GLint uknot_count,

GLfloat *uknot, GLint vknot_count, GLfloat *vknot,

*GLint u_stride, GLint v_stride, GLfloat *ctlarray,*

GLint uorder, GLint vorder, GLenum type);

Describes the vertices (or surface normals or texture coordinates) of a NURBS surface, nobj. Several of the values must be specified for both u and v parametric directions, such as the knot sequences (uknot and vknot), knot counts (uknot_count and vknot_count), and order of the polynomial (uorder and vorder) for the NURBS surface. Note that the number of control points isn't specified. Instead, it's derived by determining the number of control points along each parameter as the number of knots minus the order. Then, the number of control points for the surface is equal to the number of control points in each parametric direction, multiplied by one another. The ctlarray argument points to an array of control points. The last parameter, type, is one of the two-dimensional evaluator types. Commonly, you might use GL_MAP2_VERTEX_3 for nonrational or GL_MAP2_VERTEX_4 for rational control points, respectively. You might also use other types, such as

*GL_MAP2_TEXTURE_COORD_** or *GL_MAP2_NORMAL* to calculate and assign texture coordinates or surface normals. For example, to create a lighted (with surface normals) and textured NURBS surface, you may need to call this sequence:

```
gluBeginSurface(nobj);
gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_3);
gluEndSurface(nobj);
```

The u_stride and v_stride arguments represent the number of floating-point values between control points in each parametric direction. The evaluator type, as well as its order, affects the u_stride and v_stride values. In Example 12-5, u_stride is 12 (4 * 3) because there are three coordinates for each vertex (set by GL_MAP2_VERTEX_3) and four control points in the parametric v direction; v_stride is 3 because each vertex had three coordinates, and v control points are adjacent to one another.

Drawing a NURBS curve is similar to drawing a surface, except that all calculations are done with one parameter, *u*, rather than two. Also, for curves, **gluBeginCurve()** and **gluEndCurve()** are the bracketing routines.

void gluBeginCurve (GLUnurbsObj *nobj);

void gluEndCurve (GLUnurbsObj *nobj);

After **gluBeginCurve**(), one or more calls to **gluNurbsCurve**() define the attributes of the surface. Exactly one of these calls must have a surface type of GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4 to generate vertices. Use **gluEndCurve**() to end the definition of a surface.

void **gluNurbsCurve** (GLUnurbsObj *nobj, GLint uknot_count, GLfloat *uknot, GLint u_stride, GLfloat *ctlarray, GLint uorder, GLenum type);

Defines a NURBS curve for the object nobj. The arguments have the same meaning as those for **gluNurbsSurface()**. Note that this routine requires only one knot sequence and one declaration of the order of the NURBS object. If this curve is defined within a **gluBeginCurve()/gluEndCurve()** pair, then the type can be any of the valid one-dimensional evaluator types (such as GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4).

Trim a NURBS Surface

To create a trimmed NURBS surface with OpenGL, start as if you were creating an untrimmed surface. After calling **gluBeginSurface()** and **gluNurbsSurface()** but before calling **gluEndSurface()**, start a trim by calling **gluBeginTrim()**.

void gluBeginTrim (GLUnurbsObj *nobj);

void gluEndTrim (GLUnurbsObj *nobj);

Marks the beginning and end of the definition of a trimming loop. A trimming loop is a set of oriented, trimming curve segments (forming a closed curve) that defines the boundaries of a NURBS surface.

You can create two kinds of trimming curves, a piecewise linear curve with **gluPwlCurve(**) or a NURBS curve with **gluNurbsCurve(**). A piecewise linear curve doesn't look like what's conventionally called a curve, because it's a series of straight lines. A NURBS curve for trimming must lie within the unit square of parametric (u, v) space. The type for a NURBS trimming curve is usually GLU_MAP1_TRIM2. Less often, the type is GLU_MAP1_TRIM3, where the curve is described in a two-dimensional homogeneous space (u', v', w') by (u, v) = (u'/w', v'/w').

void gluPwlCurve (GLUnurbsObj *nobj, GLint count, GLfloat *array,

GLint stride, GLenum type);

Describes a piecewise linear trimming curve for the NURBS object nobj. There are count points on the curve, and they're given by array. The type can be either GLU_MAP1_TRIM_2 (the most common) or GLU_MAP1_TRIM_3 ((u, v, w) homogeneous parameter space). The type affects whether stride, the number of floating-point values to the next vertex, is 2 or 3.

You need to consider the orientation of trimming curves - that is, whether they're counterclockwise or clockwise - to make sure you include the desired part of the surface. If you imagine walking along a curve, everything to the left is included and everything to the right is trimmed away. For example, if your trim consists of a single counterclockwise loop, everything inside the loop is included. If the trim consists of two nonintersecting counterclockwise loops with nonintersecting interiors, everything inside either of them is included. If it consists of a counterclockwise loop with two clockwise loops inside it, the trimming region has two holes in it. The outermost trimming curve must be counterclockwise. Often, you run a trimming curve around the entire unit square to include everything within it, which is what you get by default by not specifying any trimming curves.

Trimming curves must be closed and nonintersecting. You can combine trimming curves, so long as the endpoints of the trimming curves meet to form a closed curve. You can nest curves, creating islands that float in space. Be sure to get the curve orientations right. For example, an error results if you specify a trimming region with two counterclockwise curves, one enclosed within another: The region between the curves is to the left of one and to the right of the other, so it must be both included and excluded, which is impossible. Figure 12-5 illustrates a few valid possibilities.

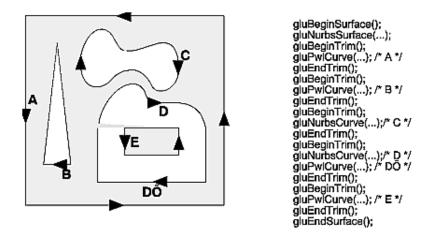


Figure 12-5 : Parametric Trimming Curves

Figure 12-6 shows the same small hill as in Figure 12-4, this time with a trimming curve that's a combination of a piecewise linear curve and a NURBS curve. The program that creates this figure is similar to that shown in Example 12-5; the differences are in the routines shown in Example 12-6.

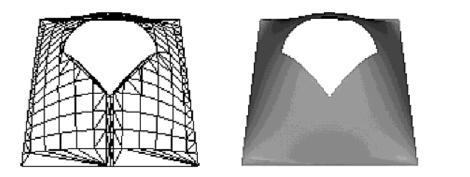


Figure 12-6 : Trimmed NURBS Surface

Example 12-6 : Trimming a NURBS Surface: trim.c

```
void display(void)
{
    GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
    GLfloat edgePt[5][2] = /* counter clockwise */
        {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0},
        {0.0, 0.0}};
GLfloat curvePt[4][2] = /* clockwise */
        {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
```

```
GLfloat curveKnots[8] =
   \{0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0\};
GLfloat pwlPt[4][2] = /* clockwise */
   \{\{0.75, 0.5\}, \{0.5, 0.25\}, \{0.25, 0.5\}\};
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glRotatef(330.0, 1.,0.,0.);
glScalef (0.5, 0.5, 0.5);
gluBeginSurface(theNurb);
gluNurbsSurface(theNurb, 8, knots, 8, knots,
                4 * 3, 3, &ctlpoints[0][0][0],
                4, 4, GL_MAP2_VERTEX_3);
gluBeginTrim (theNurb);
   gluPwlCurve (theNurb, 5, &edgePt[0][0], 2,
                GLU_MAP1_TRIM_2);
gluEndTrim (theNurb);
gluBeginTrim (theNurb);
   gluNurbsCurve (theNurb, 8, curveKnots, 2,
                  &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
   gluPwlCurve (theNurb, 3, &pwlPt[0][0], 2,
                GLU MAP1 TRIM 2);
gluEndTrim (theNurb);
gluEndSurface(theNurb);
glPopMatrix();
glFlush();
```

In Example 12-6, **gluBeginTrim**() and **gluEndTrim**() bracket each trimming curve. The first trim, with vertices defined by the array *edgePt[][]*, goes counterclockwise around the entire unit square of parametric space. This ensures that everything is drawn, provided it isn't removed by a clockwise trimming curve inside of it. The second trim is a combination of a NURBS trimming curve and a piecewise linear trimming curve. The NURBS curve ends at the points (0.9, 0.5) and (0.1, 0.5), where it is met by the piecewise linear curve, forming a closed clockwise curve.

+ +

}

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 13 Selection and Feedback

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Create applications that allow the user to select a region of the screen or pick an object drawn on the screen
- Use the OpenGL feedback mode to obtain the results of rendering calculations

Some graphics applications simply draw static images of two- and three-dimensional objects. Other applications allow the user to identify objects on the screen and then to move, modify, delete, or otherwise manipulate those objects. OpenGL is designed to support exactly such interactive applications. Since objects drawn on the screen typically undergo multiple rotations, translations, and perspective transformations, it can be difficult for you to determine which object a user is selecting in a three-dimensional scene. To help you, OpenGL provides a selection mechanism that automatically tells you which objects are drawn inside a specified region of the window. You can use this mechanism together with a special utility routine to determine which object within the region the user is specifying, or *picking*, with the cursor.

Selection is actually a mode of operation for OpenGL; feedback is another such mode. In feedback mode, you use your graphics hardware and OpenGL to perform the usual rendering calculations. Instead of using the calculated results to draw an image on the screen, however, OpenGL returns (or feeds back) the drawing information to you. For example, if you want to draw three-dimensional objects on a plotter rather than the screen, you would draw the items in feedback mode, collect the drawing instructions, and then convert them to commands the plotter can understand.

In both selection and feedback modes, drawing information is returned to the application rather than being sent to the framebuffer, as it is in rendering mode. Thus, the screen remains frozen - no drawing occurs - while OpenGL is in selection or feedback mode. In these modes, the contents of the color, depth, stencil, and accumulation buffers are not affected. This chapter explains each of these modes in its own section:

- "Selection" discusses how to use selection mode and related routines to allow a user of your application to pick an object drawn on the screen.
- "Feedback" describes how to obtain information about what would be drawn on the screen and how that information is formatted.

Selection

Typically, when you're planning to use OpenGL's selection mechanism, you first draw your scene into the framebuffer, and then you enter selection mode and redraw the scene. However, once you're in selection mode, the contents of the framebuffer don't change until you exit selection mode. When you exit selection mode, OpenGL returns a list of the primitives that intersect the viewing volume (remember that the viewing volume is defined by the current modelview and projection matrices and any additional clipping planes, as explained in Chapter 3.) Each primitive that intersects the viewing volume causes a selection *hit*. The list of primitives is actually returned as an array of integer-valued *names* and related data - the *hit records* - that correspond to the current contents of the *name stack*. You construct the name stack by loading names onto it as you issue primitive drawing commands while in selection mode. Thus, when the list of names is returned, you can use it to determine which primitives might have been selected on the screen by the user.

In addition to this selection mechanism, OpenGL provides a utility routine designed to simplify selection in some cases by restricting drawing to a small region of the viewport. Typically, you use this routine to determine which objects are drawn near the cursor, so that you can identify which object the user is picking. (You can also delimit a selection region by specifying additional clipping planes. Remember that these planes act in world space, not in screen space.) Since picking is a special case of selection, selection is described first in this chapter, and then picking.

The Basic Steps

To use the selection mechanism, you need to perform the following steps.

- 1. Specify the array to be used for the returned hit records with **glSelectBuffer**().
- 2. Enter selection mode by specifying GL_SELECT with glRenderMode().
- 3. Initialize the name stack using glInitNames() and glPushName().
- 4. Define the viewing volume you want to use for selection. Usually this is different from the viewing volume you originally used to draw the scene, so you probably want to save and then restore the current transformation state with **glPushMatrix**() and **glPopMatrix**().
- 5. Alternately issue primitive drawing commands and commands to manipulate the name stack so that each primitive of interest has an appropriate name assigned.
- 6. Exit selection mode and process the returned selection data (the hit records).

The following paragraphs describe **glSelectBuffer**() and **glRenderMode**(). In the next section, the commands to manipulate the name stack are described.

void glSelectBuffer(GLsizei size, GLuint *buffer);

Specifies the array to be used for the returned selection data. The buffer argument is a pointer to an array of unsigned integers into which the data is put, and size indicates the maximum number of values that can be stored in the array. You need to call **glSelectBuffer()** before entering selection mode.

GLint glRenderMode(GLenum mode);

Controls whether the application is in rendering, selection, or feedback mode. The mode

argument can be one of GL_RENDER (the default), GL_SELECT, or GL_FEEDBACK. The application remains in a given mode until **glRenderMode**() is called again with a different argument. Before entering selection mode, **glSelectBuffer**() must be called to specify the selection array. Similarly, before entering feedback mode, **glFeedbackBuffer**() must be called to specify the feedback array. The return value for **glRenderMode**() has meaning if the current render mode (that is, not the mode parameter) is either GL_SELECT or GL_FEEDBACK. The return value is the number of selection hits or the number of values placed in the feedback array when either mode is exited; a negative value means that the selection or feedback array has overflowed. You can use GL_RENDER_MODE with **glGetIntegerv**() to obtain the current mode.

Creating the Name Stack

As mentioned in the previous section, the name stack forms the basis for the selection information that's returned to you. To create the name stack, first initialize it with **glInitNames(**), which simply clears the stack, and then add integer names to it while issuing corresponding drawing commands. As you might expect, the commands to manipulate the stack allow you to push a name onto it (**glPushName(**)), pop a name off of it (**glPopName(**)), and replace the name on the top of the stack with a different one (**glLoadName(**)). Example 13-1 shows what your name-stack manipulation code might look like with these commands.

Example 13-1 : Creating a Name Stack

```
glInitNames();
glPushName(0);
glPushMatrix(); /* save the current transformation state */
    /* create your desired viewing volume here */
    glLoadName(1);
    drawSomeObject();
    glLoadName(2);
    drawAnotherObject();
    glLoadName(3);
    drawYetAnotherObject();
    drawJustOneMoreObject();
    drawJustOneMoreObject();
```

In this example, the first two objects to be drawn have their own names, and the third and fourth objects share a single name. With this setup, if either or both of the third and fourth objects causes a selection hit, only one hit record is returned to you. You can have multiple objects share the same name if you don't need to differentiate between them when processing the hit records.

void glInitNames(void);

Clears the name stack so that it's empty.

void glPushName(GLuint name);

Pushes name onto the name stack. Pushing a name beyond the capacity of the stack generates the error GL_STACK_OVERFLOW. The name stack's depth can vary among different OpenGL implementations, but it must be able to contain at least sixty-four names. You can use the parameter GL_NAME_STACK_DEPTH with glGetIntegerv() to obtain the depth of the name stack.

void glPopName(void);

Pops one name off the top of the name stack. Popping an empty stack generates the error

GL_STACK_UNDERFLOW.

void glLoadName(GLuint name);

Replaces the value on the top of the name stack with name. If the stack is empty, which it is right after glInitNames() is called, glLoadName() generates the error GL_INVALID_OPERATION. To avoid this, if the stack is initially empty, call glPushName() at least once to put something on the name stack before calling glLoadName().

Calls to **glPushName()**, **glPopName()**, and **glLoadName()** are ignored if you're not in selection mode. You might find that it simplifies your code to use these calls throughout your drawing code, and then use the same drawing code for both selection and normal rendering modes.

The Hit Record

In selection mode, a primitive that intersects the viewing volume causes a selection hit. Whenever a name-stack manipulation command is executed or **glRenderMode**() is called, OpenGL writes a hit record into the selection array if there's been a hit since the last time the stack was manipulated or **glRenderMode**() was called. With this process, objects that share the same name - for example, an object that's composed of more than one primitive - don't generate multiple hit records. Also, hit records aren't guaranteed to be written into the array until **glRenderMode**() is called.

Note: In addition to primitives, valid coordinates produced by **glRasterPos**() can cause a selection hit. Also, in the case of polygons, no hit occurs if the polygon would have been culled.

Each hit record consists of four items, in order.

- The number of names on the name stack when the hit occurred.
- Both the minimum and maximum window-coordinate *z* values of all vertices of the primitives that intersected the viewing volume since the last recorded hit. These two values, which lie in the range [0,1], are each multiplied by 232-1 and rounded to the nearest unsigned integer.
- The contents of the name stack at the time of the hit, with the bottommost element first.

When you enter selection mode, OpenGL initializes a pointer to the beginning of the selection array. Each time a hit record is written into the array, the pointer is updated accordingly. If writing a hit record would cause the number of values in the array to exceed the *size* argument specified with **glSelectBuffer()**, OpenGL writes as much of the record as fits in the array and sets an overflow flag. When you exit selection mode with **glRenderMode()**, this command returns the number of hit records that were written (including a partial record if there was one), clears the name stack, resets the overflow flag, and resets the stack pointer. If the overflow flag had been set, the return value is -1.

A Selection Example

In Example 13-2, four triangles (green, red, and two yellow triangles, created by calling **drawTriangle**()) and a wireframe box representing the viewing volume (**drawViewVolume**()) are drawn to the screen. Then the triangles are rendered again (**selectObjects**()), but this time in selection mode. The corresponding hit records are processed in **processHits**(), and the selection array is printed out. The first triangle generates a hit, the second one doesn't, and the third and fourth ones together generate a single hit.

Example 13-2 : Selection Example: select.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
void drawTriangle (GLfloat x1, GLfloat y1, GLfloat x2,
    GLfloat y2, GLfloat x3, GLfloat y3, GLfloat z)
ł
   glBegin (GL_TRIANGLES);
   glVertex3f (x1, y1, z);
  glVertex3f (x2, y2, z);
  glVertex3f (x3, y3, z);
  glEnd ();
}
void drawViewVolume (GLfloat x1, GLfloat x2, GLfloat y1,
                     GLfloat y2, GLfloat z1, GLfloat z2)
{
   glColor3f (1.0, 1.0, 1.0);
   glBegin (GL_LINE_LOOP);
   glVertex3f (x1, y1, -z1);
   glVertex3f (x2, y1, -z1);
   glVertex3f (x2, y2, -z1);
   glVertex3f (x1, y2, -z1);
   glEnd ();
   glBegin (GL_LINE_LOOP);
   glVertex3f (x1, y1, -z2);
   glVertex3f (x2, y1, -z2);
   glVertex3f (x2, y2, -z2);
   glVertex3f (x1, y2, -z2);
  glEnd ();
   glBegin (GL_LINES); /* 4 lines
                                         */
   glVertex3f (x1, y1, -z1);
  glVertex3f (x1, y1, -z2);
  glVertex3f (x1, y2, -z1);
  glVertex3f (x1, y2, -z2);
  glVertex3f (x2, y1, -z1);
  glVertex3f (x2, y1, -z2);
  glVertex3f (x2, y2, -z1);
  glVertex3f (x2, y2, -z2);
  glEnd ();
}
void drawScene (void)
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   gluPerspective (40.0, 4.0/3.0, 1.0, 100.0);
   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
   gluLookAt (7.5, 7.5, 12.5, 2.5, 2.5, -5.0, 0.0, 1.0, 0.0);
   glColor3f (0.0, 1.0, 0.0); /* green triangle
                                                         */
   drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
   glColor3f (1.0, 0.0, 0.0); /* red triangle
                                                         */
   drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
  glColor3f (1.0, 1.0, 0.0); /* yellow triangles
                                                         */
   drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
   drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
   drawViewVolume (0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
```

```
}
void processHits (GLint hits, GLuint buffer[])
ł
  unsigned int i, j;
  GLuint names, *ptr;
  printf ("hits = %d\n", hits);
  ptr = (GLuint *) buffer;
  for (i = 0; i < hits; i++) { /* for each hit */
     names = *ptr;
     printf (" number of names for hit = %d\n", names); ptr++;
     printf(" z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
     printf(" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
     printf (" the name is ");
      for (j = 0; j < names; j++) {</pre>
                                       /* for each name */
         printf ("%d ", *ptr); ptr++;
      }
     printf ("\n");
   }
}
#define BUFSIZE 512
void selectObjects(void)
{
  GLuint selectBuf[BUFSIZE];
  GLint hits;
  glSelectBuffer (BUFSIZE, selectBuf);
   (void) glRenderMode (GL_SELECT);
  glInitNames();
  glPushName(0);
  glPushMatrix ();
  glMatrixMode (GL PROJECTION);
  glLoadIdentity ();
  glortho (0.0, 5.0, 0.0, 5.0, 0.0, 10.0);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
  glLoadName(1);
  drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -5.0);
  glLoadName(2);
  drawTriangle (2.0, 7.0, 3.0, 7.0, 2.5, 8.0, -5.0);
  glLoadName(3);
  drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, 0.0);
  drawTriangle (2.0, 2.0, 3.0, 2.0, 2.5, 3.0, -10.0);
  glPopMatrix ();
  glFlush ();
  hits = glRenderMode (GL_RENDER);
  processHits (hits, selectBuf);
}
void init (void)
{
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_FLAT);
}
void display(void)
ł
  glClearColor (0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
drawScene ();
selectObjects ();
glFlush();
}
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (200, 200);
  glutInitWindowSize (200, 200);
  glutInitWindowPosition (100, 100);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init();
  glutDisplayFunc(display);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Picking

As an extension of the process described in the previous section, you can use selection mode to determine if objects are picked. To do this, you use a special picking matrix in conjunction with the projection matrix to restrict drawing to a small region of the viewport, typically near the cursor. Then you allow some form of input, such as clicking a mouse button, to initiate selection mode. With selection mode established and with the special picking matrix used, objects that are drawn near the cursor cause selection hits. Thus, during picking you're typically determining which objects are drawn near the cursor.

Picking is set up almost exactly like regular selection mode is, with the following major differences.

- Picking is usually triggered by an input device. In the following code examples, pressing the left mouse button invokes a function that performs picking.
- You use the utility routine **gluPickMatrix**() to multiply a special picking matrix onto the current projection matrix. This routine should be called prior to multiplying a standard projection matrix (such as **gluPerspective**() or **glOrtho**()). You'll probably want to save the contents of the projection matrix first, so the sequence of operations may look like this:

```
glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
gluPickMatrix (...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
    /* ... draw scene for picking ; perform picking ... */
glPopMatrix();
```

Another completely different way to perform picking is described in "Object Selection Using the Back Buffer" in Chapter 14. This technique uses color values to identify different components of an object.

void **gluPickMatrix**(*GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport*[4]);

Creates a projection matrix that restricts drawing to a small region of the viewport and multiplies that matrix onto the current matrix stack. The center of the picking region is (x, y) in window coordinates, typically the cursor location. width and height define the size of the picking region in screen coordinates. (You can think of the width and height as the sensitivity of the picking device.) viewport[] indicates the current viewport boundaries, which can be

glGetIntegerv(GL_VIEWPORT, GLint *viewport);

Advanced

The net result of the matrix created by gluPickMatrix() is to transform

the clipping region into the unit cube -1 ≤ (x, y, z) ≤ 1 (or -w ≤ (wx, wy, wz) ≤ w). The picking matrix effectively performs an orthogonal transformation that maps a subregion of this unit cube to the unit cube. Since the transformation is arbitrary, you can make picking work for different sorts

of regions - for example, for rotated rectangular portions of the window. In certain situations, you might find it easier to specify additional clipping planes to define the picking region.

Example 13-3 illustrates simple picking. It also demonstrates how to use multiple names to identify different components of a primitive, in this case the row and column of a selected object. A 3×3 grid of squares is drawn, with each square a different color. The board[3][3] array maintains the current amount of blue for each square. When the left mouse button is pressed, the **pickSquares**() routine is called to identify which squares were picked by the mouse. Two names identify each square in the grid - one identifies the row, and the other the column. Also, when the left mouse button is pressed, the color of all squares under the cursor position changes.

Example 13-3 : Picking Example: picksquare.c

```
#include <GL/ql.h>
#include <GL/qlu.h>
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
int board[3][3]; /* amount of color for each square */
/* Clear color value for every square on the board
                                                       */
void init(void)
{
   int i, j;
   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j ++)
         board[i][j] = 0;
   glClearColor (0.0, 0.0, 0.0, 0.0);
}
void drawSquares(GLenum mode)
ł
   GLuint i, j;
   for (i = 0; i < 3; i++) {
      if (mode == GL_SELECT)
         glLoadName (i);
      for (j = 0; j < 3; j ++) {
         if (mode == GL_SELECT)
            glPushName (j);
         glColor3f ((GLfloat) i/3.0, (GLfloat) j/3.0,
                    (GLfloat) board[i][j]/3.0);
         glRecti (i, j, i+1, j+1);
         if (mode == GL SELECT)
            glPopName ();
      }
   }
}
```

```
/* processHits prints out the contents of the
  selection array.
 *
 */
void processHits (GLint hits, GLuint buffer[])
ł
   unsigned int i, j;
   GLuint ii, jj, names, *ptr;
   printf ("hits = %d\n", hits);
  ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) { /* for each hit */
     names = *ptr;
      printf (" number of names for this hit = %d\n", names);
         ptr++;
      printf(" z1 is %g;", (float) *ptr/0x7ffffff); ptr++;
     printf(" z2 is %g\n", (float) *ptr/0x7fffffff); ptr++;
      printf (" names are ");
      for (j = 0; j < names; j++) { /* for each name */
         printf ("%d ", *ptr);
if (j == 0) /* set row and column */
            ii = *ptr;
         else if (j == 1)
            jj = *ptr;
         ptr++;
      }
     printf ("\n");
     board[ii][jj] = (board[ii][jj] + 1) % 3;
   }
}
#define BUFSIZE 512
void pickSquares(int button, int state, int x, int y)
{
   GLuint selectBuf[BUFSIZE];
   GLint hits;
   GLint viewport[4];
   if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
      return;
   glGetIntegerv (GL_VIEWPORT, viewport);
   glSelectBuffer (BUFSIZE, selectBuf);
   (void) glRenderMode (GL_SELECT);
   glInitNames();
   glPushName(0);
   glMatrixMode (GL_PROJECTION);
   glPushMatrix ();
  glLoadIdentity ();
/* create 5x5 pixel picking region near cursor location
                                                               */
   gluPickMatrix ((GLdouble) x, (GLdouble) (viewport[3] - y),
                  5.0, 5.0, viewport);
   gluOrtho2D (0.0, 3.0, 0.0, 3.0);
   drawSquares (GL_SELECT);
   glMatrixMode (GL_PROJECTION);
   glPopMatrix ();
   glFlush ();
   hits = glRenderMode (GL_RENDER);
   processHits (hits, selectBuf);
   glutPostRedisplay();
```

```
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
  drawSquares (GL_RENDER);
  glFlush();
}
void reshape(int w, int h)
  glViewport(0, 0, w, h);
  glMatrixMode(GL PROJECTION);
  glLoadIdentity();
  gluOrtho2D (0.0, 3.0, 0.0, 3.0);
  glMatrixMode(GL MODELVIEW);
  glLoadIdentity();
}
int main(int argc, char** argv)
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT SINGLE | GLUT RGB);
  glutInitWindowSize (100, 100);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
   init ();
   glutMouseFunc (pickSquares);
   glutReshapeFunc (reshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Picking with Multiple Names and a Hierarchical Model

Multiple names can also be used to choose parts of a hierarchical object in a scene. For example, if you were rendering an assembly line of automobiles, you might want the user to move the mouse to pick the third bolt on the left front tire of the third car in line. A different name can be used to identify each level of hierarchy: which car, which tire, and finally which bolt. As another example, one name can be used to describe a single molecule among other molecules, and additional names can differentiate individual atoms within that molecule.

Example 13-4 is a modification of Example 3-4, which draws an automobile with four identical wheels, each of which has five identical bolts. Code has been added to manipulate the name stack with the object hierarchy.

Example 13-4 : Creating Multiple Names

```
draw_wheel_and_bolts()
{
    long i;
    draw_wheel_body();
    for (i = 0; i < 5; i++) {
        glPushMatrix();
            glRotate(72.0*i, 0.0, 0.0, 1.0);
            glTranslatef(3.0, 0.0, 0.0);
            glPushName(i);
                 draw_bolt_body();
            glPopName();
    }
}</pre>
```

```
glPopMatrix();
    }
 }
draw_body_and_wheel_and_bolts()
{
    draw_car_body();
    glPushMatrix();
        glTranslate(40, 0, 20); /* first wheel position*/
        glPushName(1);
                                  /* name of wheel number 1 */
            draw_wheel_and_bolts();
        glPopName();
    glPopMatrix();
    glPushMatrix();
        glTranslate(40, 0, -20); /* second wheel position */
                                 /* name of wheel number 2 */
        glPushName(2);
            draw_wheel_and_bolts();
        glPopName();
    glPopMatrix();
    /* draw last two wheels similarly */
 }
```

Example 13-5 uses the routines in Example 13-4 to draw three different cars, numbered 1, 2, and 3.

Example 13-5 : Using Multiple Names

```
draw_three_cars()
ł
    glInitNames();
    glPushMatrix();
        translate_to_first_car_position();
        glPushName(1);
            draw_body_and_wheel_and_bolts();
        glPopName();
    glPopMatrix();
    glPushMatrix();
        translate_to_second_car_position();
        glPushName(2);
            draw_body_and_wheel_and_bolts();
        glPopName();
    glPopMatrix();
    glPushMatrix();
        translate_to_third_car_position();
        glPushName(3);
            draw_body_and_wheel_and_bolts();
        glPopName();
    glPopMatrix();
}
```

Assuming that picking is performed, the following are some possible name-stack return values and their interpretations. In these examples, at most one hit record is returned; also, d1 and d2 are depth values.

2 d1d2 2 1 Car 2, wheel 1

1 d1d2 3 Car 3 body

3 d1d2 1 1 0 Bolt 0 on wheel 1 on car 1

empty The pick was outside all cars

The last interpretation assumes that the bolt and wheel don't occupy the same picking region. A user might well pick both the wheel and the bolt, yielding two hits. If you receive multiple hits, you have to decide which hit to process, perhaps by using the depth values to determine which picked object is closest to the viewpoint. The use of depth values is explored further in the next section.

Picking and Depth Values

Example 13-6 demonstrates how to use depth values when picking to determine which object is picked. This program draws three overlapping rectangles in normal rendering mode. When the left mouse button is pressed, the **pickRects()** routine is called. This routine returns the cursor position, enters selection mode, initializes the name stack, and multiplies the picking matrix onto the stack before the orthographic projection matrix. A selection hit occurs for each rectangle the cursor is over when the left mouse button is clicked. Finally, the contents of the selection buffer are examined to identify which named objects were within the picking region near the cursor.

The rectangles in this program are drawn at different depth, or z, values. Since only one name is used to identify all three rectangles, only one hit can be recorded. However, if more than one rectangle is picked, that single hit has different minimum and maximum z values.

Example 13-6 : Picking with Depth Values: pickdepth.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
void init(void)
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glEnable(GL_DEPTH_TEST);
   glShadeModel(GL_FLAT);
   glDepthRange(0.0, 1.0); /* The default z mapping */
}
void drawRects(GLenum mode)
   if (mode == GL_SELECT)
      glLoadName(1);
   glBegin(GL_QUADS);
   glColor3f(1.0, 1.0, 0.0);
   glVertex3i(2, 0, 0);
   glVertex3i(2, 6, 0);
   glVertex3i(6, 6, 0);
   glVertex3i(6, 0, 0);
   glEnd();
   if (mode == GL SELECT)
      glLoadName(2);
   qlBegin(GL_QUADS);
   glColor3f(0.0, 1.0, 1.0);
   glVertex3i(3, 2, -1);
glVertex3i(3, 8, -1);
   glVertex3i(8, 8, -1);
   glVertex3i(8, 2, -1);
   glEnd();
   if (mode == GL SELECT)
      glLoadName(3);
```

```
glBegin(GL_QUADS);
   glColor3f(1.0, 0.0, 1.0);
  glVertex3i(0, 2, -2);
  glVertex3i(0, 7, -2);
  glVertex3i(5, 7, -2);
  glVertex3i(5, 2, -2);
  glEnd();
}
void processHits(GLint hits, GLuint buffer[])
{
  unsigned int i, j;
  GLuint names, *ptr;
  printf("hits = %d\n", hits);
  ptr = (GLuint *) buffer;
   for (i = 0; i < hits; i++) { /* for each hit */
     names = *ptr;
     printf(" number of names for hit = %d\n", names); ptr++;
     printf(" z1 is %g;", (float) *ptr/0x7fffffff); ptr++;
     printf(" z2 is %g\n", (float) *ptr/0x7ffffff); ptr++;
     printf("
               the name is ");
     for (j = 0; j < names; j++) { /* for each name */
        printf("%d ", *ptr); ptr++;
      }
     printf("\n");
   }
}
#define BUFSIZE 512
void pickRects(int button, int state, int x, int y)
{
  GLuint selectBuf[BUFSIZE];
  GLint hits;
  GLint viewport[4];
   if (button != GLUT_LEFT_BUTTON || state != GLUT_DOWN)
      return;
   glGetIntegerv(GL_VIEWPORT, viewport);
   glSelectBuffer(BUFSIZE, selectBuf);
   (void) glRenderMode(GL_SELECT);
  glInitNames();
  glPushName(0);
  glMatrixMode(GL PROJECTION);
  glPushMatrix();
  glLoadIdentity();
/* create 5x5 pixel picking region near cursor location */
  gluPickMatrix((GLdouble) x, (GLdouble) (viewport[3] - y),
                 5.0, 5.0, viewport);
   glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
   drawRects(GL_SELECT);
  glPopMatrix();
  glFlush();
  hits = glRenderMode(GL_RENDER);
  processHits(hits, selectBuf);
}
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
drawRects(GL RENDER);
   glFlush();
}
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 8.0, 0.0, 8.0, -0.5, 2.5);
   glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}
int main(int argc, char **argv)
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize (200, 200);
  glutInitWindowPosition (100, 100);
  glutCreateWindow(argv[0]);
   init();
  glutMouseFunc(pickRects);
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Try This

- Modify Example 13-6 to add additional calls to **glPushName**() so that multiple names are on the stack when the selection hit occurs. What will the contents of the selection buffer be?
- By default, **glDepthRange**() sets the mapping of the *z* values to [0.0,1.0]. Try modifying the **glDepthRange**() values and see how it affects the *z* values that are returned in the selection array.

Hints for Writing a Program That Uses Selection

Most programs that allow a user to interactively edit some geometry provide a mechanism for the user to pick items or groups of items for editing. For two-dimensional drawing programs (for example, text editors, page-layout programs, and circuit-design programs), it might be easier to do your own picking calculations instead of using the OpenGL picking mechanism. Often, it's easy to find bounding boxes for two-dimensional objects and to organize them in some hierarchical data structure to speed up searches. For example, picking that uses the OpenGL style in a VLSI layout program containing millions of rectangles can be relatively slow. However, using simple bounding-box information when rectangles are typically aligned with the screen could make picking in such a program extremely fast. The code is probably simpler to write, too.

As another example, since only geometric objects cause hits, you might want to create your own method for picking text. Setting the current raster position is a geometric operation, but it effectively creates only a single pickable point at the current raster position, which is typically at the lower-left corner of the text. If your editor needs to manipulate individual characters within a text string, some other picking mechanism must be used. You could draw little rectangles around each character during picking mode, but it's almost certainly easier to handle text as a special case.

If you decide to use OpenGL picking, organize your program and its data structures so that it's easy to draw appropriate lists of objects in either selection or normal drawing mode. This way, when the user picks something, you can use the same data structures for the pick operation that you use to display the items on the screen. Also, consider whether you want to allow the user to select multiple objects. One way to do this is to store a bit for each item indicating whether it's selected (however, this method requires traversing your entire list of items to find the selected items). You might find it useful to maintain a list of pointers to selected items to speed up this search. It's probably a good idea to keep the selection bit for each item as well, since when you're drawing the entire picture, you might want to draw selected items differently (for example, in a different color or with a selection box around them). Finally, consider the selection user interface. You might want to allow the user to allow the user to do the following:

- Select an item
- Sweep-select a group of items (see the next paragraphs for a description of this behavior)
- Add an item to the selection
- Add a sweep selection to the current selections
- Delete an item from a selection
- Choose a single item from a group of overlapping items

A typical solution for a two-dimensional drawing program might work as follows.

- 1. All selection is done by pointing with the mouse cursor and using the left mouse button. In what follows, *cursor* means the cursor tied to the mouse, and *button* means the left mouse button.
- 2. Clicking on an item selects it and deselects all other currently selected items. If the cursor is on top of multiple items, the smallest is selected. (In three dimensions, many other strategies work to disambiguate a selection.)
- 3. Clicking down where there is no item, holding the button down while dragging the cursor, and then releasing the button selects all the items in a screen-aligned rectangle whose corners are determined by the cursor positions when the button went down and where it came up. This is called a *sweep selection*. All items not in the swept-out region are deselected. (You must decide whether an item is selected only if it's completely within the sweep region, or if any part of it falls within the region. The completely within strategy usually works best.)
- 4. If the Shift key is held down and the user clicks on an item that isn't currently selected, that item is added to the selected list. If the clicked-upon item is selected, it's deleted from the selection list.
- 5. If a sweep selection is performed with the Shift key pressed, the items swept out are added to the current selection.
- 6. In an extremely cluttered region, it's often hard to do a sweep selection. When the button goes down, the cursor might lie on top of some item, and normally that item would be selected. You can make any operation a sweep selection, but a typical user interface interprets a

button-down on an item plus a mouse motion as a select-plus-drag operation. To solve this problem, you can have an enforced sweep selection by holding down, say, the Alt key. With this, the following set of operations constitutes a sweep selection: Alt-button down, sweep, button up. Items under the cursor when the button goes down are ignored.

- 7. If the Shift key is held during this sweep selection, the items enclosed in the sweep region are added to the current selection.
- 8. Finally, if the user clicks on multiple items, select just one of them. If the cursor isn't moved (or maybe not moved more than a pixel), and the user clicks again in the same place, deselect the item originally selected, and select a different item under the cursor. Use repeated clicks at the same point to cycle through all the possibilities.

Different rules can apply in particular situations. In a text editor, you probably don't have to worry about characters on top of each other, and selections of multiple characters are always contiguous characters in the document. Thus, you need to mark only the first and last selected characters to identify the complete selection. With text, often the best way to handle selection is to identify the positions between characters rather than the characters themselves. This allows you to have an empty selection when the beginning and end of the selection are between the same pair of characters; it also allows you to put the cursor before the first character in the document or after the final one with no special-case code.

In three-dimensional editors, you might provide ways to rotate and zoom between selections, so sophisticated schemes for cycling through the possible selections might be unnecessary. On the other hand, selection in three dimensions is difficult because the cursor's position on the screen usually gives no indication of its depth.

Feedback

Feedback is similar to selection in that once you're in either mode, no pixels are produced and the screen is frozen. Drawing does not occur; instead, information about primitives that would have been rendered is sent back to the application. The key difference between selection and feedback modes is what information is sent back. In selection mode, assigned names are returned to an array of integer values. In feedback mode, information about transformed primitives is sent back to an array of floating-point values. The values sent back to the feedback array consist of tokens that specify what type of primitive (point, line, polygon, image, or bitmap) has been processed and transformed, followed by vertex, color, or other data for that primitive. The values returned are fully transformed by lighting and viewing operations. Feedback mode is initiated by calling **glRenderMode**() with GL_FEEDBACK as the argument.

Here's how you enter and exit feedback mode.

- 1. Call **glFeedbackBuffer()** to specify the array to hold the feedback information. The arguments to this command describe what type of data and how much of it gets written into the array.
- 2. Call **glRenderMode**() with GL_FEEDBACK as the argument to enter feedback mode. (For this step, you can ignore the value returned by **glRenderMode**().) After this point, primitives aren't rasterized to produce pixels until you exit feedback mode, and the contents of the framebuffer don't change.

- 3. Draw your primitives. While issuing drawing commands, you can make several calls to **glPassThrough()** to insert markers into the returned feedback data and thus facilitate parsing.
- 4. Exit feedback mode by calling **glRenderMode**() with GL_RENDER as the argument if you want to return to normal drawing mode. The integer value returned by **glRenderMode**() is the number of values stored in the feedback array.
- 5. Parse the data in the feedback array.

void glFeedbackBuffer(GLsizei size, GLenum type, GLfloat *buffer);

Establishes a buffer for the feedback data: buffer is a pointer to an array where the data is stored. The size argument indicates the maximum number of values that can be stored in the array. The type argument describes the information fed back for each vertex in the feedback array; its possible values and their meaning are shown in Table 13-1. glFeedbackBuffer() must be called before feedback mode is entered. In the table, k is 1 in color-index mode and 4 in RGBA mode.

type Argument	Coordinates	Color	Texture	Total Values	
GL_2D	х, у	-	-	2	
GL_3D	x, y, z	-	-	3	
GL_3D_COLOR	x, y, z	k	-	3 + k	
GL_3D_COLOR_TEXTURE	x, y, z	k	4	7 + k	
GL_4D_COLOR_TEXTURE	x, y, z, w	k	4	8 + k	

 Table 13-1 : glFeedbackBuffer() type Values

The Feedback Array

In feedback mode, each primitive that would be rasterized (or each call to **glBitmap**(), **glDrawPixels**(), or **glCopyPixels**(), if the raster position is valid) generates a block of values that's copied into the feedback array. The number of values is determined by the *type* argument to **glFeedbackBuffer**(), as listed in Table 13-1. Use the appropriate value for the type of primitives you're drawing: GL_2D or GL_3D for unlit two- or three-dimensional primitives, GL_3D_COLOR for lit, three-dimensional primitives, and GL_3D_COLOR_TEXTURE or GL_4D_COLOR_TEXTURE for lit, textured, three- or four-dimensional primitives.

Each block of feedback values begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for pixel rectangles. In addition, pass-through markers that you've explicitly created can be returned in the array; the next section explains these markers in more detail. Table 13-2 shows the syntax for the feedback array; remember that the data associated with each returned vertex is as described in Table

13-1. Note that a polygon can have *n* vertices returned. Also, the *x*, *y*, *z* coordinates returned by feedback are window coordinates; if *w* is returned, it's in clip coordinates. For bitmaps and pixel rectangles, the coordinates returned are those of the current raster position. In the table, note that $GL_LINE_RESET_TOKEN$ is returned only when the line stipple is reset for that line segment.

Primitive Type	Code	Associated Data
Point	GL_POINT_TOKEN	vertex
Line	GL_LINE_TOKEN or GL_LINE_RESET_TOKEN	vertex vertex
Polygon	GL_POLYGON_TOKEN	n vertex vertex vertex
Bitmap	GL_BITMAP_TOKEN	vertex
Pixel Rectangle	GL_DRAW_PIXEL_TOKEN or GL_COPY_PIXEL_TOKEN	vertex
Pass-through	GL_PASS_THROUGH_TOKEN	a floating-point number

 Table 13-2 : Feedback Array Syntax

Using Markers in Feedback Mode

Feedback occurs after transformations, lighting, polygon culling, and interpretation of polygons by **glPolygonMode**(). It might also occur after polygons with more than three edges are broken up into triangles (if your particular OpenGL implementation renders polygons by performing this decomposition). Thus, it might be hard for you to recognize the primitives you drew in the feedback data you receive. To help parse the feedback data, call **glPassThrough**() as needed in your sequence of drawing commands to insert a marker. You might use the markers to separate the feedback values returned from different primitives, for example. This command causes GL_PASS_THROUGH_TOKEN to be written into the feedback array, followed by the floating-point value you pass in as an argument.

void glPassThrough(GLfloat token);

Inserts a marker into the stream of values written into the feedback array, if called in feedback mode. The marker consists of the code GL_PASS_THROUGH_TOKEN followed by a single floating-point value, token. This command has no effect when called outside of feedback mode. Calling glPassThrough() between glBegin() and glEnd() generates a GL_INVALID_OPERATION error.

A Feedback Example

Example 13-7 demonstrates the use of feedback mode. This program draws a lit, three-dimensional scene in normal rendering mode. Then, feedback mode is entered, and the scene is redrawn. Since the program draws lit, untextured, three-dimensional objects, the type of feedback data is GL_3D_COLOR. Since RGBA mode is used, each unclipped vertex generates seven values for the feedback buffer: *x*, *y*, *z*, *r*, *g*, *b*, and *a*.

In feedback mode, the program draws two lines as part of a line strip and then inserts a pass-through marker. Next, a point is drawn at (-100.0, -100.0, -100.0), which falls outside the orthographic viewing volume and thus doesn't put any values into the feedback array. Finally, another pass-through marker is inserted, and another point is drawn.

Example 13-7 : Feedback Mode: feedback.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
void init(void)
ł
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
}
void drawGeometry (GLenum mode)
   glBegin (GL_LINE_STRIP);
   glNormal3f (0.0, 0.0, 1.0);
   glVertex3f (30.0, 30.0, 0.0);
   glVertex3f (50.0, 60.0, 0.0);
   glVertex3f (70.0, 40.0, 0.0);
   qlEnd ();
   if (mode == GL FEEDBACK)
      glPassThrough (1.0);
   glBegin (GL_POINTS);
   glVertex3f (-100.0, -100.0, -100.0); /* will be clipped */
   glEnd ();
   if (mode == GL_FEEDBACK)
      glPassThrough (2.0);
   glBegin (GL_POINTS);
   glNormal3f (0.0, 0.0, 1.0);
   glVertex3f (50.0, 50.0, 0.0);
   glEnd ();
}
void print3DcolorVertex (GLint size, GLint *count,
                         GLfloat *buffer)
ł
   int i;
  printf (" ");
   for (i = 0; i < 7; i++) {
      printf ("%4.2f ", buffer[size-(*count)]);
      *count = *count - 1;
   }
  printf ("\n");
}
void printBuffer(GLint size, GLfloat *buffer)
ł
  GLint count;
```

```
GLfloat token;
   count = size;
   while (count)
      token = buffer[size-count]; count--;
      if (token == GL_PASS_THROUGH_TOKEN) {
         printf ("GL_PASS_THROUGH_TOKEN\n");
         printf (" %4.2f\n", buffer[size-count]);
         count--;
      }
      else if (token == GL_POINT_TOKEN) {
         printf ("GL_POINT_TOKEN\n");
         print3DcolorVertex (size, &count, buffer);
      else if (token == GL LINE TOKEN) {
         printf ("GL_LINE_TOKEN\n");
         print3DcolorVertex (size, &count, buffer);
         print3DcolorVertex (size, &count, buffer);
      }
      else if (token == GL_LINE_RESET_TOKEN) {
         printf ("GL_LINE_RESET_TOKEN\n");
         print3DcolorVertex (size, &count, buffer);
         print3DcolorVertex (size, &count, buffer);
   }
}
void display(void)
   GLfloat feedBuffer[1024];
   GLint size;
   glMatrixMode (GL PROJECTION);
  glLoadIdentity ();
  glOrtho (0.0, 100.0, 0.0, 100.0, 0.0, 1.0);
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear(GL COLOR BUFFER BIT);
  drawGeometry (GL_RENDER);
  glFeedbackBuffer (1024, GL_3D_COLOR, feedBuffer);
   (void) glRenderMode (GL_FEEDBACK);
  drawGeometry (GL_FEEDBACK);
   size = glRenderMode (GL_RENDER);
  printBuffer (size, feedBuffer);
}
int main(int argc, char** argv)
ł
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (100, 100);
  glutInitWindowPosition (100, 100);
  glutCreateWindow(argv[0]);
  init();
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Running this program generates the following output:

GL_LINE_RESET_TOKEN 30.00 30.00 0.00 0.84 0.84 0.84 1.00

```
50.00 60.00 0.00 0.84 0.84 0.84 1.00

GL_LINE_TOKEN

50.00 60.00 0.00 0.84 0.84 0.84 1.00

70.00 40.00 0.00 0.84 0.84 0.84 1.00

GL_PASS_THROUGH_TOKEN

1.00

GL_PASS_THROUGH_TOKEN

2.00

GL_POINT_TOKEN

50.00 50.00 0.00 0.84 0.84 0.84 1.00
```

Thus, the line strip drawn with these commands results in two primitives:

```
glBegin(GL_LINE_STRIP);
   glNormal3f (0.0, 0.0, 1.0);
   glVertex3f (30.0, 30.0, 0.0);
   glVertex3f (50.0, 60.0, 0.0);
   glVertex3f (70.0, 40.0, 0.0);
   glEnd();
```

The first primitive begins with GL_LINE_RESET_TOKEN, which indicates that the primitive is a line segment and that the line stipple is reset. The second primitive begins with GL_LINE_TOKEN, so it's also a line segment, but the line stipple isn't reset and hence continues from where the previous line segment left off. Each of the two vertices for these lines generates seven values for the feedback array. Note that the RGBA values for all four vertices in these two lines are (0.84, 0.84, 0.84, 1.0), which is a very light gray color with the maximum alpha value. These color values are a result of the interaction of the surface normal and lighting parameters.

Since no feedback data is generated between the first and second pass-through markers, you can deduce that any primitives drawn between the first two calls to **glPassThrough()** were clipped out of the viewing volume. Finally, the point at (50.0, 50.0, 0.0) is drawn, and its associated data is copied into the feedback array.

Note: In both feedback and selection modes, information on objects is returned prior to any fragment tests. Thus, objects that would not be drawn due to failure of the scissor, alpha, depth, or stencil tests may still have their data processed and returned in both feedback and selection modes.

Try This

Make changes to Example 13-7 and see how they affect the feedback values that are returned. For example, change the coordinate values of **glOrtho(**). Change the lighting variables, or eliminate lighting altogether and change the feedback type to GL_3D. Or add more primitives to see what other geometry (such as filled polygons) contributes to the feedback array.



OpenGL Programming Guide (Addison-Wesley Publishing Company)

Chapter 14 Now That You Know

Chapter Objectives

This chapter doesn't have objectives in the same way that previous chapters do. It's simply a collection of topics that describe ideas you might find useful for your application. Some topics, such as error handling, don't fit into other categories, but are too short for an entire chapter.

OpenGL is kind of a bag of low-level tools; now that you know about those tools, you can use them to implement higher-level functions. This chapter presents several examples of such higher-level capabilities.

This chapter discusses a variety of techniques based on OpenGL commands that illustrate some of the not-so-obvious uses to which you can put these commands. The examples are in no particular order and aren't related to each other. The idea is to read the section headings and skip to the examples that you find interesting. For your convenience, the headings are listed and explained briefly here.

Note: Most of the examples in the rest of this guide are complete and can be compiled and run as is. In this chapter, however, there are no complete programs, and you have to do a bit of work on your own to make them run.

- "Error Handling" tells you how to check for OpenGL error conditions.
- "Which Version Am I Using?" describes how to find out details about the implementation, including the version number. This can be useful for writing applications that are backward compatible with earlier versions of OpenGL.
- "Extensions to the Standard" presents techniques to identify and use vendor-specific extensions to the OpenGL standard.
- "Cheesy Translucency" explains how to use polygon stippling to achieve translucency; this is particularly useful when you don't have blending hardware available.
- "An Easy Fade Effect" shows how to use polygon stippling to create the effect of a fade into the background.
- "Object Selection Using the Back Buffer" describes how to use the back buffer in a double-buffered system to handle simple object picking.
- "Cheap Image Transformation" discusses how to draw a distorted version of a bitmapped image by drawing each pixel as a quadrilateral.

- "Displaying Layers" explains how to display multiple different layers of materials and indicate where the materials overlap.
- "Antialiased Characters" describes how to draw smoother fonts.
- "Drawing Round Points" describes how to draw near-round points.
- "Interpolating Images" shows how to smoothly blend from one image to the another.
- "Making Decals" explains how to draw two images, where one is a sort of decal that should always appear on top of the other.
- "Drawing Filled, Concave Polygons Using the Stencil Buffer" tells you how to draw concave polygons, nonsimple polygons, and polygons with holes by using the stencil buffer.
- "Finding Interference Regions" describes how to determine where three-dimensional pieces overlap.
- "Shadows" describes how to draw shadows of lit objects.
- "Hidden-Line Removal" discusses how to draw a wireframe object with hidden lines removed by using the stencil buffer.
- "Texture-Mapping Applications" describes several clever uses for texture mapping, such as rotating and warping images.
- "Drawing Depth-Buffered Images" tells you how to combine images in a depth-buffered environment.
- "Dirichlet Domains" explains how to find the Dirichlet domain of a set of points using the depth buffer.
- "Life in the Stencil Buffer" explains how to implement the Game of Life using the stencil buffer.
- "Alternative Uses for glDrawPixels() and glCopyPixels()" describes how to use these two commands for such effects as fake video, airbrushing, and transposed images.

Error Handling

The truth is, your program will make mistakes. Use of error-handling routines are essential during development and are highly recommended for commercially released applications. (Unless you can give a 100% guarantee your program will never generate an OpenGL error condition. Get real!) OpenGL has simple error-handling routines for the base GL and GLU libraries.

When OpenGL detects an error (in either the base GL or GLU), it records a current error code. The command that caused the error is ignored, so it has no effect on OpenGL state or on the framebuffer contents. (If the error recorded was GL_OUT_OF_MEMORY, however, the results of the command are undefined.) Once recorded, the current error code isn't cleared - that is, additional

errors aren't recorded - until you call the query command **glGetError**(), which returns the current error code. After you've queried and cleared the current error code, or if there's no error to begin with, **glGetError**() returns GL_NO_ERROR.

GLenum glGetError(void);

Returns the value of the error flag. When an error occurs in either the GL or GLU, the error flag is set to the appropriate error code value. If GL_NO_ERROR is returned, there has been no detectable error since the last call to glGetError(), or since the GL was initialized. No other errors are recorded until glGetError() is called, the error code is returned, and the flag is reset to GL_NO_ERROR.

It is strongly recommended that you call **glGetError**() at least once in each **display**() routine. Table 14-1 lists the basic defined OpenGL error codes.

Error Code	Description
GL_INVALID_ENUM	GLenum argument out of range
GL_INVALID_VALUE	Numeric argument out of range
GL_INVALID_OPERATION	Operation illegal in current state
GL_STACK_OVERFLOW	Command would cause a stack overflow
GL_STACK_UNDERFLOW	Command would cause a stack underflow
GL_OUT_OF_MEMORY	Not enough memory left to execute command

 Table 14-1 : OpenGL Error Codes

There are also thirty-seven GLU NURBS errors (with non-descriptive constant names, GLU_NURBS_ERROR1, GLU_NURBS_ERROR2, and so on), fourteen tessellator errors (GLU_TESS_MISSING_BEGIN_POLYGON, GLU_TESS_MISSING_END_POLYGON, GLU_TESS_MISSING_BEGIN_CONTOUR, GLU_TESS_MISSING_END_CONTOUR, GLU_TESS_COORD_TOO_LARGE, GLU_TESS_NEED_COMBINE_CALLBACK, and eight generically named GLU_TESS_ERROR*), and GLU_INCOMPATIBLE_GL_VERSION. Also, the GLU defines the error codes GLU_INVALID_ENUM, GLU_INVALID_VALUE, and GLU_OUT_OF_MEMORY, which have the same meaning as the related OpenGL codes.

To obtain a printable, descriptive string corresponding to either a GL or GLU error code, use the GLU routine **gluErrorString**().

const GLubyte* gluErrorString(GLenum errorCode);

Returns a pointer to a descriptive string that corresponds to the OpenGL or GLU error number passed in errorCode.

In Example 14-1, a simple error handling routine is shown.

Example 14-1 : Querying and Printing an Error

```
GLenum errCode;
const GLubyte *errString;
if ((errCode = glGetError()) != GL_NO_ERROR) {
    errString = gluErrorString(errCode);
    fprintf (stderr, "OpenGL Error: %s\n", errString);
}
```

Note: The string returned by gluErrorString() must not be altered or freed by the application.

Which Version Am I Using?

The portability of OpenGL applications is one of OpenGL's attractive features. However, new versions of OpenGL introduce new features, which may introduce backward compatibility problems. In addition, you may want your application to perform equally well on a variety of implementations. For example, you might make texture mapping the default rendering mode on one machine, but only have flat shading on another. You can use **glGetString()** to obtain release information about your OpenGL implementation.

const GLubyte* glGetString(GLenum name); Returns a pointer to a string that describes an aspect of the OpenGL implementation. name can be one of the following: GL_VENDOR, GL_RENDERER, GL_VERSION, or GL_EXTENSIONS.

GL_VENDOR returns the name of the company responsible for the OpenGL implementation. GL_RENDERER returns an identifier of the renderer, which is usually the hardware platform. For more about GL_EXTENSIONS, see the next section, "Extensions to the Standard."

GL_VERSION returns a string that identifies the version number of this implementation of OpenGL. The version string is laid out as follows:

<version number><space><vendor-specific information>

The version number is either of the form

major_number.minor_number

or

major_number.minor_number.release_number

where the numbers all have one or more digits. The vendor-specific information is optional. For example, if this OpenGL implementation is from the fictitious XYZ Corporation, the string returned might be

1.1.4 XYZ-OS 3.2

which means that this implementation is XYZ's fourth release of an OpenGL library that conforms to the specification for OpenGL Version 1.1. It probably also means this is release 3.2 of XYZ's

proprietary operating system.

Another way to query the version number for OpenGL is to look for the symbolic constant (use the preprocessor statement #ifdef) named GL_VERSION_1_1. The absence of the constant GL_VERSION_1_1 means that you have OpenGL Version 1.0.

Note: If running from client to server, such as when performing indirect rendering with the OpenGL extension to the X Window System, the client and server may be different versions. If your client version is ahead of your server, your client might request an operation that is not supported on your server.

Utility Library Version

gluGetString() is a query function for the Utility Library (GLU) and is similar to glGetString().

const GLubyte* gluGetString(GLenum name); Returns a pointer to a string that describes an aspect of the OpenGL implementation. name can be one of the following: GLU_VERSION, or GLU_EXTENSIONS.

Note that **gluGetString()** was not available in GLU 1.0. Another way to query the version number for GLU is to look for the symbolic constant GLU_VERSION_1_1. The absence of the constant GLU_VERSION_1_1 means that you have GLU 1.0.

Extensions to the Standard

OpenGL has a formal written specification that describes what operations comprise the library. An individual vendor or a group of vendors may decide to include additional functionality to their released implementation.

New routine and symbolic constant names clearly indicate whether a feature is part of the OpenGL standard or a vendor-specific extension. To make a vendor-specific name, the vendor appends a company identifier (in uppercase) and, if needed, additional information, such as a machine name. For example, if XYZ Corporation wants to add a new routine and symbolic constant, they might be of the form **glCommandXYZ**() and GL_DEFINITION_XYZ. If XYZ Corporation wants to have an extension that is available only on its FooBar graphics board, then the names might be **glCommandXYZfb**() and GL_DEFINITION_XYZ_FB.

If two of more vendors agree to implement the same extension, then the procedures and constants are suffixed with the more generic EXT (**glCommandEXT**() and GL_DEFINITION_EXT).

If you want to know if a particular extension is supported on your implementation, use **glGetString**(GL_EXTENSIONS). This returns a list of all the extensions in the implementation, separated by spaces. If you want to find out if a specific extension is supported, use the code in Example 14-2 to search through the list and match the extension name. Return GL_TRUE, if it is; GL_FALSE, if it isn't.

Example 14-2 : Find Out If An Extension Is Supported

```
static GLboolean QueryExtension(char *extName)
{
```

```
char *p = (char *) glGetString(GL_EXTENSIONS);
char *end = p + strlen(p);
while (p < end) {
    int n = strcspn(p, " ");
        if ((strlen(extName)==n) && (strncmp(extName,p,n)==0)) {
        return GL_TRUE;
    }
    p += (n + 1);
}
return GL_FALSE;
}
```

Cheesy Translucency

You can use polygon stippling to simulate a translucent material. This is an especially good solution for systems that don't have blending hardware. Since polygon stipple patterns are 32x32 bits, or 1024 bits, you can go from opaque to transparent in 1023 steps. (In practice, that's many more steps than you need!) For example, if you want a surface that lets through 29 percent of the light, simply make up a stipple pattern where 29 percent (roughly 297) of the pixels in the mask are zero and the rest are one. Even if your surfaces have the same translucency, don't use the same stipple pattern for each one, as they cover exactly the same bits on the screen. Make up a different pattern for each by randomly selecting the appropriate number of pixels to be zero. (See "Displaying Points, Lines, and Polygons" in Chapter 2 for more information about polygon stippling.)

If you don't like the effect with random pixels turned on, you can use regular patterns, but they don't work as well when transparent surfaces are stacked. This is often not a problem because most scenes have relatively few translucent regions that overlap. In a picture of an automobile with translucent windows, your line of sight can go through at most two windows, and usually it's only one.

An Easy Fade Effect

Suppose you have an image that you want to fade gradually to some background color. Define a series of polygon stipple patterns, each of which has more bits turned on so that they represent denser and denser patterns. Then use these patterns repeatedly with a polygon large enough to cover the region over which you want to fade. For example, suppose you want to fade to black in 16 steps. First define 16 different pattern arrays:

```
GLubyte stips[16][4*32];
```

Then load them in such a way that each has one-sixteenth of the pixels in a 32×32 stipple pattern turned on and that the bitwise OR of all the stipple patterns is all ones. After that, the following code does the trick:

```
draw_the_picture();
glColor3f(0.0, 0.0, 0.0); /* set color to black */
for (i = 0; i < 16; i++) {
    glPolygonStipple(&stips[i][0]);
    draw_a_polygon_large_enough_to_cover_the_whole_region();
}
```

In some OpenGL implementations, you might get better performance by first compiling the stipple patterns into display lists. During your initialization, do something like this:

```
#define STIP_OFFSET 100
for (i = 0; i < 16; i++) {
   glNewList(i+STIP_OFFSET, GL_COMPILE);
   glPolygonStipple(&stips[i][0]);
   glEndList();
}</pre>
```

Then, replace this line in the first code fragment

```
glPolygonStipple(&stips[i][0]);
```

with

glCallList(i);

By compiling the command to set the stipple into a display list, OpenGL might be able to rearrange the data in the *stips[][]* array into the hardware-specific form required for maximum stipple-setting speed.

Another application for this technique is if you're drawing a changing picture and want to leave some blur behind that gradually fades out to give some indication of past motion. For example, suppose you're simulating a planetary system and you want to leave trails on the planets to show a recent portion of their path. Again, assuming you want to fade in sixteen steps, set up the stipple patterns as before (using the display-list version, say), and have the main simulation loop look something like this:

Each time through the loop, you clear one-sixteenth of the pixels. Any pixel that hasn't had a planet on it for sixteen frames is certain to be cleared to black. Of course, if your system supports blending in hardware, it's easier to blend in a certain amount of background color with each frame. (See "Displaying Points, Lines, and Polygons" in Chapter 2 for polygon stippling details, Chapter 7 for more information about display lists, and "Blending" in Chapter 6 for information about blending.)

Object Selection Using the Back Buffer

Although the OpenGL selection mechanism (see "Selection" in Chapter 13) is powerful and flexible, it can be cumbersome to use. Often, the situation is simple: Your application draws a scene composed of a substantial number of objects; the user points to an object with the mouse, and the application needs to find the item under the tip of the cursor.

One way to do this requires your application to be running in double-buffer mode. When the user picks an object, the application redraws the entire scene in the back buffer, but instead of using the normal colors for objects, it encodes some kind of object identifier for each object's color. The

application then simply reads back the pixel under the cursor, and the value of that pixel encodes the number of the picked object. If many picks are expected for a single, static picture, you can read the entire color buffer once and look in your copy for each attempted pick, rather than read back each pixel individually.

Note that this scheme has an advantage over standard selection in that it picks the object that's in front if multiple objects appear at the same pixel, one behind the other. Since the image with false colors is drawn in the back buffer, the user never sees it; you can redraw the back buffer (or copy it from the front buffer) before swapping the buffers. In color-index mode, the encoding is simple - send the object identifier as the index. In RGBA mode, encode the bits of the identifier into the R, G, and B components.

Be aware that you can run out of identifiers if there are too many objects in the scene. For example, suppose you're running in color-index mode on a system that has 4-bit buffers for color-index information (16 possible different indices) in each of the color buffers, but the scene has thousands of pickable items. To address this issue, the picking can be done in a few passes. To think about this in concrete terms, assume there are fewer than 4096 items, so all the object identifiers can be encoded in 12 bits. In the first pass, draw the scene using indices composed of the 4 high-order bits, then use the second and third passes to draw the middle 4 bits and the 4 low-order bits. After each pass, read the pixel under the cursor, extract the bits, and pack them together at the end to get the object identifier.

With this method, the picking takes three times as long, but that's often acceptable. Note that after you have the high-order 4 bits, you eliminate 15/16 of all objects, so you really need to draw only 1/16 of them for the second pass. Similarly, after the second pass, 255 of the 256 possible items have been eliminated. The first pass thus takes about as long as drawing a single frame does, but the second and third passes can be up to 16 and 256 times as fast.

If you're trying to write portable code that works on different systems, break up your object identifiers into chunks that fit on the lowest common denominator of those systems. Also, keep in mind that your system might perform automatic dithering in RGB mode. If this is the case, turn off dithering.

Cheap Image Transformation

If you want to draw a distorted version of a bitmapped image (perhaps simply stretched or rotated, or perhaps drastically modified by some mathematical function), there are many possibilities. You can use the image as a texture map, which allows you to scale, rotate, or otherwise distort the image. If you just want to scale the image, you can use **glPixelZoom**().

In many cases, you can achieve good results by drawing the image of each pixel as a quadrilateral. Although this scheme doesn't produce images that are as nice as those you would get by applying a sophisticated filtering algorithm (and it might not be sufficient for sophisticated users), it's a lot quicker.

To make the problem more concrete, assume that the original image is *m* pixels by *n* pixels, with coordinates chosen from $[0, m-1] \times [0, n-1]$. Let the distortion functions be x(m,n) and y(m,n). For example, if the distortion is simply a zooming by a factor of 3.2, then x(m,n) = 3.2*m and y(m,n) = 3.2*n. The following code draws the distorted image:

```
glShadeModel(GL_FLAT);
glScale(3.2, 3.2, 1.0);
for (j=0; j < n; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i <= m; i++) {
        glVertex2i(i,j);
        glVertex2i(i, j+1);
        set_color(i,j);
    }
    glEnd();
}
```

This code draws each transformed pixel in a solid color equal to that pixel's color and scales the image size by 3.2. The routine **set_color()** stands for whatever the appropriate OpenGL command is to set the color of the image pixel.

The following is a slightly more complex version that distorts the image using the functions x(i,j) and y(i,j):

```
glShadeModel(GL_FLAT);
for (j=0; j < n; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i <= m; i++) {
        glVertex2i(x(i,j), y(i,j));
        glVertex2i(x(i,j+1), y(i,j+1));
        set_color(i,j);
    }
    glEnd();
}
```

An even better distorted image can be drawn with the following code:

```
glShadeModel(GL_SMOOTH);
for (j=0; j < (n-1); j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i < m; i++) {
        set_color(i,j);
        glVertex2i(x(i,j), y(i,j));
        set_color(i,j+1);
        glVertex2i(x(i,j+1), y(i,j+1));
    }
    glEnd();
}
```

This code smoothly interpolates color across each quadrilateral. Note that this version produces one fewer quadrilateral in each dimension than do the flat-shaded versions, because the color image is being used to specify colors at the quadrilateral vertices. In addition, you can antialias the polygons with the appropriate blending function (GL_SRC_ALPHA, GL_ONE) to get an even nicer image.

Displaying Layers

In some applications such as semiconductor layout programs, you want to display multiple different layers of materials and indicate where the materials overlap each other.

As a simple example, suppose you have three different substances that can be layered. At any point, eight possible combinations of layers can occur, as shown in Table 14-2.

	Layer 1	Layer 2	Layer 3	Color	
0	absent	absent	absent	black	
1	present	absent	absent	red	
2	absent	present	absent	green	
3	present	present	absent	blue	
4	absent	absent	present	pink	
5	present	absent	present	yellow	
6	absent	present	present	white	
7	present	present	present	gray	

Table 14-2 : Eight Combinations of Layers

You want your program to display eight different colors, depending on the layers present. One arbitrary possibility is shown in the last column of the table. To use this method, use color-index mode and load your color map so that entry 0 is black, entry 1 is red, entry 2 is green, and so on. Note that if the numbers from 0 through 7 are written in binary, the 4 bit is turned on whenever layer 3 appears, the 2 bit whenever layer 2 appears, and the 1 bit whenever layer 1 appears.

To clear the window, set the writemask to 7 (all three layers) and set the clearing color to 0. To draw your image, set the color to 7, and then when you want to draw something in layer n, set the writemask to n. In other types of applications, it might be necessary to selectively erase in a layer, in which case you would use the writemasks just discussed, but set the color to 0 instead of 7. (See "Masking Buffers" in Chapter 10 for more information about writemasks.)

Antialiased Characters

Using the standard technique for drawing characters with **glBitmap**(), drawing each pixel of a character is an all-or-nothing affair - the pixel is either turned on or not. If you're drawing black characters on a white background, for example, the resulting pixels are either black or white, never a shade of gray. Much smoother, higher-quality images can be achieved if intermediate colors are used when rendering characters (grays, in this example).

Assuming that you're drawing black characters on a white background, imagine a highly magnified picture of the pixels on the screen, with a high-resolution character outline superimposed on it, as shown in the left side of Figure 14-1.

1993		000000	100000		 000000	85555		101010101	101010101	1816181
									All and a second	
	7									
53535"										395951
				A			90 1 93400			

0	1	1	1	1	1	1	1	1	1	1	1
2	3	2	2	2	3	3	2	2	_2_	3	2
2	Λ	0	0	0	3	3	0	0	0	1	3
2/	0	0	0	0	3	3	0	0	0	0	2
Ū	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0	0	0	0	0
0	0	0	0	0	3	3	0)	0	0	0	0
0	0	0	1-	-2	3	3	2	-1-	0	0	0

Figure 14-1 : Antialiased Characters

Notice that some of the pixels are completely enclosed by the character's outline and should be painted black; some pixels are completely outside the outline and should be painted white; but many pixels should ideally be painted some shade of gray, where the darkness of the gray corresponds to the amount of black in the pixel. If this technique is used, the resulting image on the screen looks better.

If speed and memory usage are of no concern, each character can be drawn as a small image instead of as a bitmap. If you're using RGBA mode, however, this method might require up to 32 bits per pixel of the character to be stored and drawn, instead of the 1 bit per pixel in a standard character. Alternatively, you could use one 8-bit index per pixel and convert these indices to RGBA by table lookup during transfer. In many cases, a compromise is possible that allows you to draw the character with a few gray levels between black and white (say, two or three), and the resulting font description requires only 2 or 3 bits per pixel of storage.

The numbers in the right side of Figure 14-1 indicate the approximate percentage coverage of each pixel: 0 means approximately empty, 1 means approximately one-third coverage, 2 means two-thirds, and 3 means completely covered. If pixels labeled 0 are painted white, pixels labeled 3 are painted black, and pixels labeled 1 and 2 are painted one-third and two-thirds black, respectively, the resulting character looks quite good. Only 2 bits are required to store the numbers 0, 1, 2, and 3, so for 2 bits per pixel, four levels of gray can be saved.

There are basically two methods to implement antialiased characters, depending on whether you're in RGBA or color-index mode.

In RGBA mode, define three different character bitmaps, corresponding to where 1, 2, and 3 appear in Figure 14-1. Set the color to white, and clear for the background. Set the color to one-third gray (RGB = (0.666, 0.666, 0.666)), and draw all the pixels with a 1 in them. Then set RGB = (0.333, 0.333), draw with the 2 bitmap, and use RGB = (0.0, 0.0, 0.0) for the 3 bitmap. What you're

doing is defining three different fonts and redrawing the string three times, where each pass fills in the bits of the appropriate color densities.

In color-index mode, you can do exactly the same thing, but if you're willing to set up the color map correctly and use writemasks, you can get away with only two bitmaps per character and two passes per string. In the preceding example, set up one bitmap that has a 1 wherever 1 or 3 appears in the character. Set up a second bitmap that has a 1 wherever a 2 or a 3 appears. Load the color map so that 0 gives white, 1 gives light gray, 2 gives dark gray, and 3 gives black. Set the color to 3 (11 in binary) and the writemask to 1, and draw the first bitmap. Then change the writemask to 2, and draw the second. Where 0 appears in Figure 14-1, nothing is drawn in the framebuffer. Where 1, 2, and 3 appear, 1, 2, and 3 appear in the framebuffer.

For this example with only four gray levels, the savings is small - two passes instead of three. If eight gray levels were used instead, the RGBA method would require seven passes, and the color-map masking technique would require only three. With sixteen gray levels, the comparison is fifteen passes to four passes. (See "Masking Buffers" in Chapter 10 for more information about writemasks and "Bitmaps and Fonts" in Chapter 8 for more information about drawing bitmaps.)

Try This

• Can you see how to do RGBA rendering using no more images than the optimized color-index case? Hint: How are RGB fragments normally merged into the color buffer when antialiasing is desired?

Drawing Round Points

Draw near-round, aliased points by enabling point antialiasing, turning blending off, and using an alpha function that passes only fragments with alpha greater than 0.5. (See "Antialiasing" and "Blending" in Chapter 6 for more information about these topics.)

Interpolating Images

Suppose you have a pair of images (where *image* can mean a bitmap image, or a picture generated using geometry in the usual way), and you want to smoothly blend from one to the other. This can be done easily using the alpha component and appropriate blending operations. Let's say you want to accomplish the blending in ten steps, where image A is shown in frame 0 and image B is shown in frame 9. The obvious approach is to draw image A with alpha equal to (9-&igr;)/9 and image B with an alpha of i/9 in frame i.

The problem with this method is that both images must be drawn in each frame. A faster approach is to draw image A in frame 0. To get frame 1, blend in 1/9 of image B and 8/9 of what's there. For frame 2, blend in 1/8 of image B with 7/8 of what's there. For frame 3, blend in 1/7 of image B with 6/7 of what's there, and so on. For the last step, you're just drawing 1/1 of image B blended with 0/1 of what's left, yielding image B exactly.

To see that this works, if for frame *i* you have

 $\frac{(9-i)\,A}{9}+\frac{iB}{9}$

and you blend in B/(9- &igr;) with (8- &igr;)/(9- &igr;) of what's there, you get

$$\frac{B}{9-i} + \frac{8-i}{9-i} \left[\frac{(9-i)A}{9} + \frac{iB}{9} \right] = \frac{9-(i+1)A}{9} + \frac{(i+1)B}{9}$$

(See "Blending" in Chapter 6.)

Making Decals

Suppose you're drawing a complex three-dimensional picture using depth-buffering to eliminate the hidden surfaces. Suppose further that one part of your picture is composed of coplanar figures A and B, where B is a sort of decal that should always appear on top of figure A.

Your first approach might be to draw B after you've drawn A, setting the depth-buffering function to replace on greater or equal. Due to the finite precision of the floating-point representations of the vertices, however, round-off error can cause polygon B to be sometimes a bit in front and sometimes a bit behind figure A. Here's one solution to this problem.

- 1. Disable the depth buffer for writing, and render A.
- 2. Enable the depth buffer for writing, and render B.
- 3. Disable the color buffer for writing, and render A again.
- 4. Enable the color buffer for writing.

Note that during the entire process, the depth-buffer test is enabled. In step 1, A is rendered wherever it should be, but none of the depth-buffer values are changed; thus, in step 2, wherever B appears over A, B is guaranteed to be drawn. Step 3 simply makes sure that all of the depth values under A are updated correctly, but since RGBA writes are disabled, the color pixels are unaffected. Finally, step 4 returns the system to the default state (writing is enabled both in the depth buffer and in the color buffer).

If a stencil buffer is available, the following simpler technique works.

- 1. Configure the stencil buffer to write one if the depth test passes, and zero otherwise. Render A.
- 2. Configure the stencil buffer to make no stencil value change, but to render only where stencil values are one. Disable the depth-buffer test and its update. Render B.

With this method, it's not necessary to initialize the contents of the stencil buffer at any time, because the stencil value of all pixels of interest (that is, those rendered by A) are set when A is rendered. Be sure to reenable the depth test and disable the stencil test before additional polygons

Drawing Filled, Concave Polygons Using the Stencil Buffer

Consider the concave polygon 1234567 shown in Figure 14-2. Imagine that it's drawn as a series of triangles: 123, 134, 145, 156, 167, all of which are shown in the figure. The heavier line represents the original polygon boundary. Drawing all these triangles divides the buffer into nine regions A, B, C, ..., I, where region I is outside all the triangles.

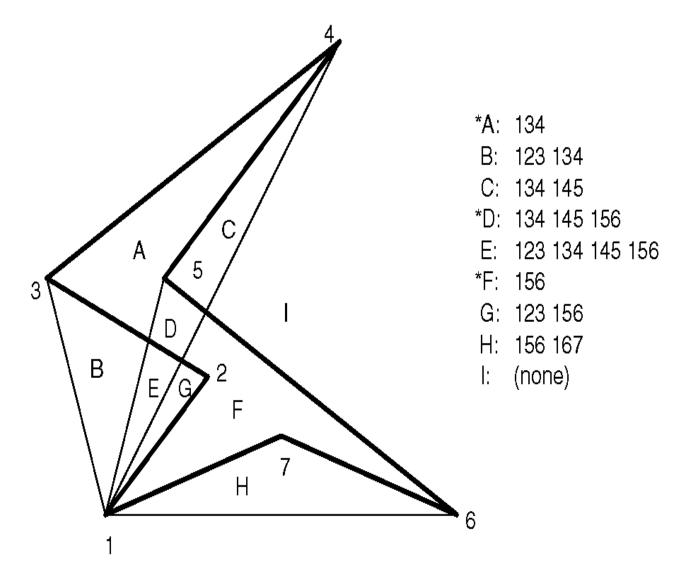


Figure 14-2 : Concave Polygon

In the text of the figure, each of the region names is followed by a list of the triangles that cover it. Regions A, D, and F make up the original polygon; note that these three regions are covered by an odd number of triangles. Every other region is covered by an even number of triangles (possibly zero). Thus, to render the inside of the concave polygon, you just need to render regions that are enclosed by an odd number of triangles. This can be done using the stencil buffer, with a two-pass algorithm.

First, clear the stencil buffer and disable writing into the color buffer. Next, draw each of the triangles in turn, using the GL_INVERT function in the stencil buffer. (For best performance, use triangle fans.) This flips the value between zero and a nonzero value every time a triangle is drawn that covers a pixel. After all the triangles are drawn, if a pixel is covered an even number of times, the value in the stencil buffers is zero; otherwise, it's nonzero. Finally, draw a large polygon over the whole region (or redraw the triangles), but allow drawing only where the stencil buffer is nonzero.

Note: There's a slight generalization of the preceding technique, where you don't need to start with a polygon vertex. In the 1234567 example, let P be any point on or off the polygon. Draw the triangles: P12, P23, P34, P45, P56, P67, and P71. Regions covered by an odd number of triangles are inside; other regions are outside. This is a generalization in that if P happens to be one of the polygon's edges, one of the triangles is empty.

This technique can be used to fill both nonsimple polygons (polygons whose edges cross each other) and polygons with holes. The following example illustrates how to handle a complicated polygon with two regions, one four-sided and one five-sided. Assume further that there's a triangular and a four-sided hole (it doesn't matter in which regions the holes lie). Let the two regions be abcd and efghi, and the holes jkl and mnop. Let z be any point on the plane. Draw the following triangles:

zab zbc zcd zda zef zfg zgh zhi zie zjk zkl zlj zmn zno zop zpm

Mark regions covered by an odd number of triangles as *in*, and those covered by an even number as *out*. (See "Stencil Test" in Chapter 10 for more information about the stencil buffer.)

Finding Interference Regions

If you're designing a mechanical part made from smaller three-dimensional pieces, you often want to display regions where the pieces overlap. In many cases, such regions indicate design errors where parts of a machine interfere with each other. In the case of moving parts, it can be even more valuable, since a search for interfering regions can be done through a complete mechanical cycle of the design. The method for doing this is complicated, and the description here might be too brief. Complete details can be found in the paper *Interactive Inspection of Solids: Cross-sections and Interferences*, by Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider (SIGGRAPH 1992 Proceedings).

The method is related to the capping algorithm described in "Stencil Test" in Chapter 10. The idea is to pass an arbitrary clipping plane through the objects that you want to test for interference, and then determine when a portion of the clipping plane is inside more than one object at a time. For a static image, the clipping plane can be moved manually to highlight interfering regions; for a dynamic image, it might be easier to use a grid of clipping planes to search for all possible interferences.

Draw each of the objects you want to check and clip them against the clipping plane. Note which pixels are inside the object at that clipping plane using an odd-even count in the stencil buffer, as explained in the preceding section. (For properly formed objects, a point is inside the object if a ray drawn from that point to the eye intersects an odd number of surfaces of the object.) To find

interferences, you need to find pixels in the framebuffer where the clipping plane is in the interior of two or more regions at once; in other words, in the intersection of the interiors of any pair of objects.

If multiple objects need to be tested for mutual intersection, store 1 bit every time some intersection appears, and another bit wherever the clipping buffer is inside any of the objects (the union of the objects' interiors). For each new object, determine its interior, find the intersection of that interior with the union of the interiors of the objects so far tested, and keep track of the intersection points. Then add the interior points of the new object to the union of the objects' interiors.

You can perform the operations described in the preceding paragraph by using different bits in the stencil buffer together with various masking operations. Three bits of stencil buffer are required per pixel - one for the toggling to determine the interior of each object, one for the union of all interiors discovered so far, and one for the regions where interference has occurred so far. To make this discussion more concrete, assume the 1 bit of the stencil buffer is for toggling interior/exterior, the 2 bit is the running union, and the 4 bit is for interferences so far. For each object that you're going to render, clear the 1 bit (using a stencil mask of one and clearing to zero), then toggle the 1 bit by keeping the stencil mask as one and using the GL_INVERT stencil operation.

You can find intersections and unions of the bits in the stencil buffers using the stenciling operations. For example, to make bits in buffer 2 be the union of the bits in buffers 1 and 2, mask the stencil to those 2 bits, and draw something over the entire object with the stencil function set to pass if anything nonzero occurs. This happens if the bits in buffer 1, buffer 2, or both are turned on. If the comparison succeeds, write a 1 in buffer 2. Also, make sure that drawing in the color buffer is disabled. An intersection calculation is similar - set the function to pass only if the value in the two buffers is equal to 3 (bits turned on in both buffers 1 and 2). Write the result into the correct buffer. (See "Stencil Test" in Chapter 10.)

Shadows

Every possible projection of three-dimensional space to three-dimensional space can be achieved with a suitable 4×4 invertible matrix and homogeneous coordinates. If the matrix isn't invertible but has rank 3, it projects three-dimensional space onto a two-dimensional plane. Every such possible projection can be achieved with a suitable rank-3 4×4 matrix. To find the shadow of an arbitrary object on an arbitrary plane from an arbitrary light source (possibly at infinity), you need to find a matrix representing that projection, multiply it on the matrix stack, and draw the object in the shadow color. Keep in mind that you need to project onto each plane that you're calling the "ground."

As a simple illustration, assume the light is at the origin, and the equation of the ground plane is ax+by+c+d=0. Given a vertex S=(sx,sy,sz,1), the line from the light through S includes all points & agr; S, where & agr; is an arbitrary real number. The point where this line intersects the plane occurs when

&agr; (a*sz+b*sy+c*sz) + d = 0,

so

&agr; = - & dgr; /(a*sx+b*sy+c*sz).

Plugging this back into the line, we get

- &dgr; (&sgr; &xgr; , &sgr; &psgr; , &sgr; &zgr;)/(&agr; * &sgr; &xgr; + &bgr; * &sgr; &psgr; + &khgr; * &sgr; &zgr;)

for the point of intersection.

The matrix that maps S to this point for every S is

 $\begin{bmatrix} -d & 0 & 0 & a \\ 0 & -d & 0 & b \\ 0 & 0 & -d & c \\ 0 & 0 & 0 & 0 \end{bmatrix}$

This matrix can be used if you first translate the world so that the light is at the origin.

If the light is from an infinite source, all you have is a point S and a direction D = (dx, dy, dz). Points along the line are given by

S + &agr; D

Proceeding as before, the intersection of this line with the plane is given by

a(sx+&agr; dx)+b(sy+&agr; dy)+c(sz+&agr; dz)+d=0

Solving for &agr; , plugging that back into the equation for a line, and then determining a projection matrix gives

```
\begin{bmatrix} b^*dy + c^*dz & -a^*dy & -a^*dz & 0\\ -b^*dx & a^*dx + c^*dz & -b^*dz & 0\\ -c^*dx & -c^*dy & a^*dx + b^*dy & 0\\ -d^*dx & -d^*dy & -d^*dz & a^*dx + b^*dy^*c^*dz \end{bmatrix}
```

This matrix works given the plane and an arbitrary direction vector. There's no need to translate anything first. (See Chapter 3 and Appendix F.)

Hidden-Line Removal

If you want to draw a wireframe object with hidden lines removed, one approach is to draw the outlines using lines and then fill the interiors of the polygons making up the surface with polygons having the background color. With depth-buffering enabled, this interior fill covers any outlines that would be obscured by faces closer to the eye. This method would work, except that there's no guarantee that the interior of the object falls entirely inside the polygon's outline; in fact, it might overlap it in various places.

There's an easy, two-pass solution using either polygon offset or the stencil buffer. Polygon offset is usually the preferred technique, since polygon offset is almost always faster than stencil buffer. Both methods are described here, so you can see how both approaches to the problem work.

Hidden-Line Removal with Polygon Offset

To use polygon offset to accomplish hidden-line removal, the object is drawn twice. The highlighted edges are drawn in the foreground color, using filled polygons but with the polygon mode GL_LINE to rasterize it as a wireframe. Then the filled polygons are drawn with the default polygon mode, which fills the interior of the wireframe, and with enough polygon offset to nudge the filled polygons a little farther from the eye. With the polygon offset, the interior recedes just enough that the highlighted edges are drawn without unpleasant visual artifacts.

```
glEnable(GL_DEPTH_TEST);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
set_color(foreground);
draw_object_with_filled_polygons();
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glPolygonOffset(1.0, 1.0);
set_color(background);
draw_object_with_filled_polygons();
glDisable(GL_POLYGON_OFFSET_FILL);
```

You may need to adjust the amount of offset needed (for wider lines, for example). (See "Polygon Offset" in Chapter 6 for more information.)

Hidden-Line Removal with the Stencil Buffer

Using the stencil buffer for hidden-line removal is a more complicated procedure. For each polygon, you'll need to clear the stencil buffer, and then draw the outline both in the framebuffer and in the stencil buffer. Then when you fill the interior, enable drawing only where the stencil buffer is still clear. To avoid doing an entire stencil-buffer clear for each polygon, an easy way to clear it is simply to draw 0's into the buffer using the same polygon outline. In this way, you need to clear the entire stencil buffer only once.

For example, the following code represents the inner loop you might use to perform such hidden-line removal. Each polygon is outlined in the foreground color, filled with the background color, and then outlined again in the foreground color. The stencil buffer is used to keep the fill color of each polygon from overwriting its outline. To optimize performance, the stencil and color parameters are changed only twice per loop by using the same values both times the polygon outline is drawn.

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 1);
glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
set_color(foreground);
for (i=0; i < max; i++) {
    outline_polygon(i);
    set_color(background);
    glStencilFunc(GL_EQUAL, 0, 1);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
    fill_polygon(i);
```

```
set_color(foreground);
glStencilFunc(GL_ALWAYS, 0, 1);
glStencilOp(GL_INVERT, GL_INVERT, GL_INVERT);
outline_polygon(i);
}
```

```
(See "Stencil Test" in Chapter 10.)
```

Texture-Mapping Applications

Texture mapping is quite powerful, and it can be used in some interesting ways. Here are a few advanced applications of texture mapping.

- Antialiased text Define a texture map for each character at a relatively high resolution, and then map them onto smaller areas using the filtering provided by texturing. This also makes text appear correctly on surfaces that aren't aligned with the screen, but are tilted and have some perspective distortion.
- Antialiased lines These can be done like antialiased text: Make the line in the texture several pixels wide, and use the texture filtering to antialias the lines.
- Image scaling and rotation If you put an image into a texture map and use that texture to map onto a polygon, rotating and scaling the polygon effectively rotates and scales the image.
- Image warping As in the preceding example, store the image as a texture map, but map it to some spline-defined surface (use evaluators). As you warp the surface, the image follows the warping.
- Projecting images Put the image in a texture map, and project it as a spotlight, creating a slide projector effect. (See "The q Coordinate" in Chapter 9 for more information about how to model a spotlight using textures.)

(See Chapter 3 for information about rotating and scaling, Chapter 9 for more information about creating textures, and Chapter 12 for details on evaluators.)

Drawing Depth-Buffered Images

For complex static backgrounds, the rendering time for the geometric description of the background can be greater than the time it takes to draw a pixel image of the rendered background. If there's a fixed background and a relatively simple changing foreground, you may want to draw the background and its associated depth-buffered version as an image rather than render it geometrically. The foreground might also consist of items that are time-consuming to render, but whose framebuffer images and depth buffers are available. You can render these items into a depth-buffered environment using a two-pass algorithm.

For example, if you're drawing a model of a molecule made of spheres, you might have an image of a beautifully rendered sphere and its associated depth-buffer values that were calculated using Phong shading or ray-tracing or by using some other scheme that isn't directly available through

OpenGL. To draw a complex model, you might be required to draw hundreds of such spheres, which should be depth-buffered together.

To add a depth-buffered image to the scene, first draw the image's depth-buffer values into the depth buffer using **glDrawPixels**(). Then enable depth-buffering, set the writemask to zero so that no drawing occurs, and enable stenciling such that the stencil buffers get drawn whenever a write to the depth buffer occurs.

Then draw the image into the color buffer, masked by the stencil buffer you've just written so that writing occurs only when there's a 1 in the stencil buffer. During this write, set the stenciling function to zero out the stencil buffer so that it's automatically cleared when it's time to add the next image to the scene. If the objects are to be moved nearer to or farther from the viewer, you need to use an orthographic projection; in these cases, you use GL_DEPTH_BIAS with **glPixelTransfer***() to move the depth image. (See "Coordinate System Survival Kit" in Chapter 2, "Depth Test" and "Stencil Test" in Chapter 10, and Chapter 8 for details on **glDrawPixels**() and **glPixelTransfer***().)

Dirichlet Domains

Given a set S of points on a plane, the Dirichlet domain or Voronoi polygon of one of the points is the set of all points in the plane closer to that point than to any other point in the set S. These points provide the solution to many problems in computational geometry. Figure 14-3 shows outlines of the Dirichlet domains for a set of points.

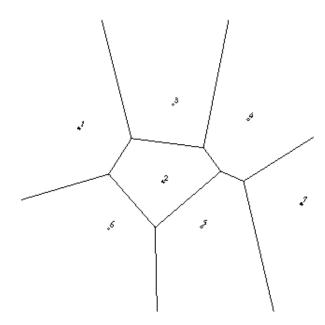


Figure 14-3 : Dirichlet Domains

If you draw a depth-buffered cone with its apex at the point in a different color than each of the points in S, the Dirichlet domain for each point is drawn in that color. The easiest way to do this is to precompute a cone's depth in an image and use the image as the depth-buffer values as described in the preceding section. You don't need an image to draw in the framebuffer as in the case of shaded spheres, however. While you're drawing into the depth buffer, use the stencil buffer to

record the pixels where drawing should occur by first clearing it and then writing nonzero values wherever the depth test succeeds. To draw the Dirichlet region, draw a polygon over the entire window, but enable drawing only where the stencil buffers are nonzero.

You can do this perhaps more easily by rendering cones of uniform color with a simple depth buffer, but a good cone might require thousands of polygons. The technique described in this section can render much higher-quality cones much more quickly. (See "A Hidden-Surface Removal Survival Kit" in Chapter 5 and "Depth Test" in Chapter 10.)

Life in the Stencil Buffer

The Game of Life, invented by John Conway, is played on a rectangular grid where each grid location is "alive" or "dead." To calculate the next generation from the current one, count the number of live neighbors for each grid location (the eight adjacent grid locations are neighbors). A grid location is alive in generation n+1 if it was alive in generation n and has exactly two or three live neighbors, or if it was dead in generation n and has exactly three live neighbors. In all other cases, it is dead in generation n+1. This game generates some incredibly interesting patterns given different initial configurations. (See Martin Gardner, "Mathematical Games," *Scientific American*, vol. 223, no. 4, October 1970, p. 120-123.) Figure 14-4 shows six generations from a game.

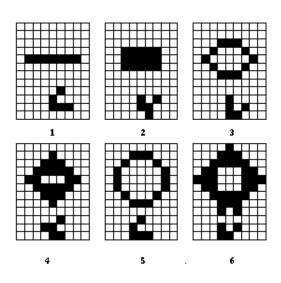


Figure 14-4 : Six Generations from the Game of Life

One way to create this game using OpenGL is to use a multipass algorithm. Keep the data in the color buffer, one pixel for each grid point. Assume that black (all zeros) is the background color, and the color of a live pixel is nonzero. Initialize by clearing the depth and stencil buffers to zero, set the depth-buffer writemask to zero, and set the depth comparison function so that it passes on not-equal. To iterate, read the image off the screen, enable drawing into the depth buffer, and set the stencil function so that it increments whenever a depth comparison succeeds but leaves the stencil buffer unchanged otherwise. Disable drawing into the color buffer.

Next, draw the image eight times, offset one pixel in each vertical, horizontal, and diagonal direction. When you're done, the stencil buffer contains a count of the number of live neighbors for

each pixel. Enable drawing to the color buffer, set the color to the color for live cells, and set the stencil function to draw only if the value in the stencil buffer is 3 (three live neighbors). In addition, if this drawing occurs, decrement the value in the stencil buffer. Then draw a rectangle covering the image; this paints each cell that has exactly three live neighbors with the "alive" color.

At this point, the stencil buffers contain 0, 1, 2, 4, 5, 6, 7, 8, and the values under the 2's are correct. The values under 0, 1, 4, 5, 6, 7, and 8 must be cleared to the "dead" color. Set the stencil function to draw whenever the value is not 2, and to zero the stencil values in all cases. Then draw a large polygon of the "dead" color across the entire image. You're done.

For a usable demonstration program, you might want to zoom the grid up to a size larger than a single pixel; it's hard to see detailed patterns with a single pixel per grid point. (See "Coordinate System Survival Kit" in Chapter 2, and "Depth Test" and "Stencil Test" in Chapter 10.)

Alternative Uses for glDrawPixels() and glCopyPixels()

You might think of **glDrawPixels**() as a way to draw a rectangular region of pixels to the screen. Although this is often what it's used for, some other interesting uses are outlined here.

- Video Even if your machine doesn't have special video hardware, you can display short movie clips by repeatedly drawing frames with **glDrawPixels**() in the same region of the back buffer and then swapping the buffers. The size of the frames you can display with reasonable performance using this method depends on your hardware's drawing speed, so you might be limited to 100 × 100 pixel movies (or smaller) if you want smooth fake video.
- Airbrush In a paint program, your airbrush (or paintbrush) shape can be simulated using alpha values. The color of the paint is represented as the color values. To paint with a circular brush in blue, repeatedly draw a blue square with **glDrawPixels**() where the alpha values are largest in the center and taper to zero at the edges of a circle centered in the square. Draw using a blending function that uses alpha of the incoming color and (1-alpha) of the color already at the pixel. If the alpha values in the brush are all much less than one, you have to paint over an area repeatedly to get a solid color. If the alpha values are near one, each brush stroke pretty much obliterates the colors underneath.
- Filtered Zooms If you zoom a pixel image by a nonintegral amount, OpenGL effectively uses a box filter, which can lead to rather severe aliasing effects. To improve the filtering, jitter the resulting image by amounts less than a pixel and redraw it multiple times, using alpha blending to average the resulting pixels. The result is a filtered zoom.
- Transposing Images You can swap same-size images in place with **glCopyPixels**() using the XOR operation. With this method, you can avoid having to read the images back into processor memory. If A and B represent the two images, the operation looks like this:
 - 1. A = A XOR B
 - 2. B = A XOR B
 - 3. A = A XOR B



OpenGL Programming Guide (Addison-Wesley Publishing Company)

Appendix A Order of Operations

This book describes all the operations performed between when vertices are initially specified and fragments are finally written into the framebuffer. The chapters of this book are arranged in an order that facilitates learning rather than in the exact order in which these operations are actually performed. Sometimes the exact order of operations doesn't matter - for example, surfaces can be converted to polygons and then transformed, or transformed first and then converted to polygons, with identical results - and different implementations of OpenGL might do things differently.

This appendix describes a possible order; any implementation is required to give equivalent results. If you want more details than are presented here, see the *OpenGL Reference Manual*.

This appendix has the following major sections:

- "Overview"
- "Geometric Operations"
- "Pixel Operations"
- "Fragment Operations"
- "Odds and Ends"

Overview

This section gives an overview of the order of operations, as shown in Figure A-1. Geometric data (vertices, lines, and polygons) follows the path through the row of boxes that include evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) is treated differently for part of the process. Both types of data undergo the rasterization and per-fragment operations before the final pixel data is written into the framebuffer.

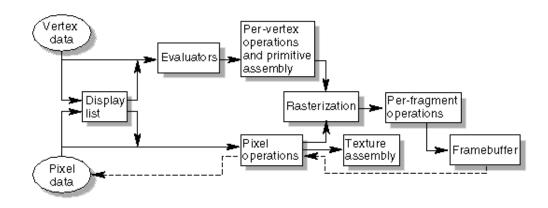


Figure A-1 : Order of Operations

All data, whether it describes geometry or pixels, can be saved in a display list or processed immediately. When a display list is executed, the data is sent from the display list just as if it were sent by the application.

All geometric primitives are eventually described by vertices. If evaluators are used, that data is converted to vertices and treated as vertices from then on. Vertex data may also be stored in and used from specialized vertex arrays. Per-vertex calculations are performed on each vertex, followed by rasterization to fragments. For pixel data, pixel operations are performed, and the results are either stored in the texture memory, used for polygon stippling, or rasterized to fragments.

Finally, the fragments are subjected to a series of per-fragment operations, after which the final pixel values are drawn into the framebuffer.

Geometric Operations

Geometric data, whether it comes from a display list, an evaluator, the vertices of a rectangle, or as raw data, consists of a set of vertices and the type of primitive it describes (a vertex, line, or polygon). Vertex data includes not only the (x, y, z, w) coordinates, but also a normal vector, texture coordinates, a RGBA color, a color index, material properties, and edge-flag data. All these elements except the vertex's coordinates can be specified in any order, and default values exist as well. As soon as the vertex command **glVertex***() is issued, the components are padded, if necessary, to four dimensions (using z = 0 and w = 1), and the current values of all the elements are associated with the vertex. The complete set of vertex data is then processed. (If vertex arrays are used, vertex data may be batch processed and processed vertices may be reused.)

Per-Vertex Operations

In the per-vertex operations stage of processing, each vertex's spatial coordinates are transformed by the modelview matrix, while the normal vector is transformed by that matrix's inverse transpose and renormalized if specified. If automatic texture generation is enabled, new texture coordinates are generated from the transformed vertex coordinates, and they replace the vertex's old texture coordinates. The texture coordinates are then transformed by the current texture matrix and passed on to the primitive assembly step. Meanwhile, the lighting calculations, if enabled, are performed using the transformed vertex and normal vector coordinates, and the current material, lights, and lighting model. These calculations generate new colors or indices that are clamped or masked to the appropriate range and passed on to the primitive assembly step.

Primitive Assembly

Primitive assembly differs, depending on whether the primitive is a point, a line, or a polygon. If flat shading is enabled, the colors or indices of all the vertices in a line or polygon are set to the same value. If special clipping planes are defined and enabled, they're used to clip primitives of all three types. (The clipping-plane equations are transformed by the inverse transpose of the modelview matrix when they're specified.) Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending on how the line or polygon is clipped. After this clipping, the spatial coordinates of each vertex are transformed by the projection matrix, and the results are clipped against the standard viewing planes $x = \pm \&ohgr;$, $y = \pm \&ohgr;$, and $z = \pm \&ohgr;$.

If selection is enabled, any primitive not eliminated by clipping generates a selection-hit report, and no further processing is performed. Without selection, perspective division by *w* occurs and the viewport and depth-range operations are applied. Also, if the primitive is a polygon, it's then subjected to a culling test (if culling is enabled). A polygon might convert to vertices or lines, depending on the polygon mode.

Finally, points, lines, and polygons are rasterized to fragments, taking into account polygon or line stipples, line width, and point size. Rasterization involves determining which squares of an integer grid in window coordinates are occupied by the primitive. If antialiasing is enabled, coverage (the portion of the square that is occupied by the primitive) is also computed. Color and depth values are also assigned to each such square. If polygon offset is enabled, depth values are slightly modified by a calculated offset value.

Pixel Operations

Pixels from host memory are first unpacked into the proper number of components. The OpenGL unpacking facility handles a number of different formats. Next, the data is scaled, biased, and processed using a pixel map. The results are clamped to an appropriate range depending on the data type and then either written in the texture memory for use in texture mapping or rasterized to fragments.

If pixel data is read from the framebuffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. The results are packed into an appropriate format and then returned to processor memory.

The pixel copy operation is similar to a combination of the unpacking and transfer operations, except that packing and unpacking is unnecessary, and only a single pass is made through the transfer operations before the data is written back into the framebuffer.

Texture Memory

OpenGL Version 1.1 provides additional control over texture memory. Texture image data can be

specified from framebuffer memory, as well as processor memory. All or a portion of a texture image may be replaced. Texture data may be stored in texture objects, which can be loaded into texture memory. If there are too many texture objects to fit into texture memory at the same time, the textures that have the highest priorities remain in the texture memory.

Fragment Operations

If texturing is enabled, a texel is generated from texture memory for each fragment and applied to the fragment. Then fog calculations are performed, if they're enabled, followed by the application of coverage (antialiasing) values, if antialiasing is enabled.

Next comes scissoring, followed by the alpha test (in RGBA mode only), the stencil test, and the depth-buffer test. If in RGBA mode, blending is performed. Blending is followed by dithering and logical operation. All these operations may be disabled.

The fragment is then masked by a color mask or an index mask, depending on the mode, and drawn into the appropriate buffer. If fragments are being written into the stencil or depth buffer, masking occurs after the stencil and depth tests, and the results are drawn into the framebuffer without performing the blending, dithering, or logical operation.

Odds and Ends

Matrix operations deal with the current matrix stack, which can be the modelview, the projection, or the texture matrix stack. The commands **glMultMatrix***(), **glLoadMatrix***(), and **glLoadIdentity**() are applied to the top matrix on the stack, while **glTranslate***(), **glRotate***(), **glScale***(), **glOrtho**(), and **glFrustum**() are used to create a matrix that's multiplied by the top matrix. When the modelview matrix is modified, its inverse transpose is also generated for normal vector transformation.

The commands that set the current raster position are treated exactly like a vertex command up until when rasterization would occur. At this point, the value is saved and is used in the rasterization of pixel data.

The various **glClear()** commands bypass all operations except scissoring, dithering, and writemasking.



OpenGL Programming Guide (Addison-Wesley Publishing Company)

Appendix B State Variables

This appendix lists the queryable OpenGL state variables, their default values, and the commands for obtaining the values of these variables. The *OpenGL Reference Manual* contains detailed information on all the commands and constants discussed in this appendix. This appendix has these major sections:

- "The Query Commands"
- "OpenGL State Variables"

The Query Commands

In addition to the basic commands to obtain the values of simple state variables (commands such as **glGetIntegerv()** and **glIsEnabled()**, which are described in "Basic State Management" in Chapter 2), there are other specialized commands to return more complex state variables. The prototypes for these specialized commands are listed here. Some of these routines, such as **glGetError()** and **glGetString()**, have been discussed in more detail elsewhere in the book.

To find out when you need to use these commands and their corresponding symbolic constants, use the tables in the next section, "OpenGL State Variables." Also see the *OpenGL Reference Manual*.

void glGetClipPlane(GLenum plane, GLdouble *equation);

GLenum **glGetError**(void);

void glGetLight{if}v(GLenum light, GLenum pname, TYPE *params);

void glGetMap{ifd}v(GLenum target, GLenum query, TYPE *v);

void glGetMaterial{if}v(GLenum face, GLenum pname, TYPE * params);

void glGetPixelMap{f ui us}v(GLenum map, TYPE *values);

void glGetPolygonStipple(GLubyte *mask);

const GLubyte * glGetString(GLenum name);

void glGetTexEnv{if}v(GLenum target, GLenum pname, TYPE *params);

void glGetTexGen{ifd}v(GLenum coord, GLenum pname, TYPE *params);

void glGetTexImage(GLenum target, GLint level, GLenum format, GLenum type, GLvoid *pixels);

void glGetTexLevelParameter{if}v(GLenum target, GLint level, GLenum pname, TYPE *params);

void glGetTexParameter{if}v(GLenum target, GLenum pname, TYPE *params);

void gluGetNurbsProperty(GLUnurbsObj *nobj, GLenum property, GLfloat *value);

```
const GLubyte * gluGetString(GLenum name);
```

void gluGetTessProperty(GLUtesselator *tess, GLenum which, GLdouble *data);

OpenGL State Variables

The following pages contain tables that list the names of queryable state variables. For each variable, the tables list a description of it, its attribute group, its initial or minimum value, and the suggested **glGet***() command to use for obtaining it. State variables that can be obtained using **glGetBooleanv**(), **glGetIntegerv**(), **glGetFloatv**(), or **glGetDoublev**() are listed with just one of these commands - the one that's most appropriate given the type of data to be returned. (Some vertex array variables can be queried only with **glGetPointerv**().) These state variables can't be obtained using **glIsEnabled**(). However, state variables for which **glIsEnabled**() is listed as the query command can also be obtained using **glGetBooleanv**(), **glGetFloatv**(), **glGetFloatv**(), and **glGetDoublev**(). State variables for which any other command is listed as the query command can be obtained.

If one or more attribute groups are listed, the state variable belongs to the listed group or groups. If no attribute group is listed, the variable doesn't belong to any group. **glPushAttrib()**, **glPopAttrib()**, and **glPopClientAttrib()** may be used to save and restore all state values that belong to an attribute group. (See "Attribute Groups" in Chapter 2 for more information.)

All queryable state variables, except the implementation-dependent ones, have initial values. If no initial value is listed, you need to consult either the section where that variable is discussed or the *OpenGL Reference Manual* to determine its initial value.

Current Values and Associated Data

Table B-1 : State Variables for Current Values and Associated Data

State Variable	Description	Attribute Group	Initial Value
GL_CURRENT_COLOR	Current color	current	1, 1, 1, 1
GL_CURRENT_INDEX	Current color index	current	1
GL_CURRENT_TEXTURE_COORDS	Current texture coordinates	current	0, 0, 0, 1
GL_CURRENT_NORMAL	Current normal	current	0, 0, 1
GL_CURRENT_RASTER_POSITION	Current raster position	current	0, 0, 0, 1
GL_CURRENT_RASTER_DISTANCE	Current raster distance	current	0
GL_CURRENT_RASTER_COLOR	Color associated with raster position	current	1, 1, 1, 1
GL_CURRENT_RASTER_INDEX	Color index associated with raster position	current	1
GL_CURRENT_RASTER_TEXTURE_COORDS	Texture coordinates associated with raster position	current	0, 0, 0, 1
GL_CURRENT_RASTER_POSITION_VALID	Raster position valid bit	current	GL_TRUE
GL_EDGE_FLAG	Edge flag	current	GL_TRUE

State Variable	Description	Attribute Group	Initial Value
GL_VERTEX_ARRAY	Vertex array enable	vertex-array	GL_FALSE
GL_VERTEX_ARRAY_SIZE	Coordinates per vertex	vertex-array	4
GL_VERTEX_ARRAY_TYPE	Type of vertex coordinates	vertex-array	GL_FLOAT
GL_VERTEX_ARRAY_STRIDE	Stride between vertices	vertex-array	0
GL_VERTEX_ARRAY_POINTER	Pointer to the vertex array	vertex-array	NULL
GL_NORMAL_ARRAY	Normal array enable	vertex-array	GL_FALSE
GL_NORMAL_ARRAY_TYPE	Type of normal coordinates	vertex-array	GL_FLOAT
GL_NORMAL_ARRAY_STRIDE	Stride between normals	vertex-array	0
GL_NORMAL_ARRAY_POINTER	Pointer to the normal array	vertex-array	NULL
GL_COLOR_ARRAY	RGBA color array enable	vertex-array	GL_FALSE
GL_COLOR_ARRAY_SIZE	Colors per vertex	vertex-array	4

 Table B-2 : (continued) Vertex Array State Variables

	זר	1	
GL_COLOR_ARRAY_TYPE	Type of color components	vertex-array	GL_FLOAT
GL_COLOR_ARRAY_STRIDE	Stride between colors	vertex-array	0
GL_COLOR_ARRAY_POINTER	Pointer to the color array	vertex-array	NULL
GL_INDEX_ARRAY	Color-index array enable	vertex-array	GL_FALSE
GL_INDEX_ARRAY_TYPE	Type of color indices	vertex-array	GL_FLOAT
GL_INDEX_ARRAY_STRIDE	Stride between color indices	vertex-array	0
GL_INDEX_ARRAY_POINTER	Pointer to the index array	vertex-array	NULL
GL_TEXTURE_COORD_ARRAY	Texture coordinate array enable	vertex-array	GL_FALSE
GL_TEXTURE_COORD_ARRAY_SIZE	Texture coordinates per element	vertex-array	4
GL_TEXTURE_COORD_ARRAY_TYPE	Type of texture coordinates	vertex-array	GL_FLOAT
GL_TEXTURE_COORD_ARRAY_STRIDE	Stride between texture coordinates	vertex-array	0

GL_TEXTURE_COORD_ARRAY_POINTER	Pointer to the texture coordinate array	vertex-array	NULL
GL_EDGE_FLAG_ARRAY	Edge flag array enable	vertex-array	GL_FALSE
GL_EDGE_FLAG_ARRAY_STRIDE	Stride between edge flags	vertex-array	0
GL_EDGE_FLAG_ARRAY_POINTER	Pointer to the edge flag array	vertex-array	NULL

Transformation

State Variable	Description	Attribute Group	Initial Value	
GL_MODELVIEW_MATRIX	Modelview matrix stack	-	Identity	٤
GL_PROJECTION_MATRIX	Projection matrix stack	-	Identity	٤
GL_TEXTURE_MATRIX	Texture matrix stack	-	Identity	£
GL_VIEWPORT	Viewport origin and extent	viewport	-	Ę
GL_DEPTH_RANGE	Depth range near and far	viewport	0, 1	Ę
GL_MODELVIEW_STACK_DEPTH	Modelview matrix stack pointer	-	1	<u></u>

Table B-3 : Transformation State Variables

GL_PROJECTION_STACK_DEPTH	Projection matrix stack pointer	-	1	ž
GL_TEXTURE_STACK_DEPTH	Texture matrix stack pointer	-	1	Ę
GL_MATRIX_MODE	Current matrix mode	transform	GL_MODELVIEW	£
GL_NORMALIZE	Current normal normalization on/off	transform/ enable	GL_FALSE	ž
GL_CLIP_PLANE <i>i</i>	User clipping plane coefficients	transform	0, 0, 0, 0	£
GL_CLIP_PLANE <i>i</i>	<i>i</i> th user clipping plane enabled	transform/ enable	GL_FALSE	£

Coloring

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_FOG_COLOR	Fog color	fog	0, 0, 0, 0	glGetFloatv()
GL_FOG_INDEX	Fog index	fog	0	glGetFloatv()
GL_FOG_DENSITY	Exponential fog density	fog	1.0	glGetFloatv()
GL_FOG_START	Linear fog start	fog	0.0	glGetFloatv()
GL_FOG_END	Linear fog end	fog	1.0	glGetFloatv()
GL_FOG_MODE	Fog mode	fog	GL_EXP	glGetIntegerv()
GL_FOG	True if fog enabled	fog/enable	GL_FALSE	glIsEnabled()
GL_SHADE_MODEL	glShadeModel() setting	lighting	GL_SMOOTH	glGetIntegerv()

Lighting

See also Table 5-1 and Table 5-3 for initial values.

State Variable	Description	Attribute Group	Initial Value	(
GL_LIGHTING	True if lighting is enabled	lighting/e nable	GL_FALSE	g
GL_COLOR_MATERIAL	True if color tracking is enabled	lighting	GL_FALSE	g

GL_COLOR_MATERIAL_PARAMETER	Material properties tracking current color	lighting	GL_AMBIENT_ AND_DIFFUSE	g
GL_COLOR_MATERIAL_FACE	Face(s) affected by color tracking	lighting	GL_FRONT_ AND_BACK	g
GL_AMBIENT	Ambient material color	lighting	(0.2, 0.2, 0.2, 1.0)	g
GL_DIFFUSE	Diffuse material color	lighting	(0.8, 0.8, 0.8, 1.0)	g
GL_SPECULAR	Specular material color	lighting	(0.0, 0.0, 0.0, 1.0)	g
GL_EMISSION	Emissive material color	lighting	(0.0, 0.0, 0.0, 1.0)	g
GL_SHININESS	Specular exponent of material	lighting	0.0	g
GL_LIGHT_MODEL_AMBIENT	Ambient scene color	lighting	(0.2, 0.2, 0.2, 1.0)	g
GL_LIGHT_MODEL_LOCAL_VIEWER	Viewer is local	lighting	GL_FALSE	g
GL_LIGHT_MODEL_TWO_SIDE	Use two-sided lighting	lighting	GL_FALSE	g
GL_AMBIENT	Ambient intensity of light <i>i</i>	lighting	(0.0,0.0,0.0,1.0)	g

GL_DIFFUSE	Diffuse intensity of light <i>i</i>	lighting	-	g
GL_SPECULAR	Specular intensity of light <i>i</i>	lighting	-	g
GL_POSITION	Position of light <i>i</i>	lighting	(0.0, 0.0, 1.0, 0.0)	g
GL_CONSTANT_ATTENUATION	Constant attenuation factor	lighting	1.0	g
GL_LINEAR_ATTENUATION	Linear attenuation factor	lighting	0.0	g
GL_QUADRATIC_ATTENUATION	Quadratic attenuation factor	lighting	0.0	g
GL_SPOT_DIRECTION	Spotlight direction of light <i>i</i>	lighting	(0.0, 0.0, -1.0)	g
GL_SPOT_EXPONENT	Spotlight exponent of light <i>i</i>	lighting	0.0	g
GL_SPOT_CUTOFF	Spotlight angle of light <i>i</i>	lighting	180.0	g
GL_LIGHTi	True if light <i>i</i> enabled	lighting/e nable	GL_FALSE	g
GL_COLOR_INDEXES	ca, cd, and cs for color-index lighting	lighting/e nable	0, 1, 1	g

Rasterization

State Variable	Description	Attribute Group	Initial Value
GL_POINT_SIZE	Point size	point	1.0
GL_POINT_SMOOTH	Point antialiasing on	point/enable	GL_FALSE
GL_LINE_WIDTH	Line width	line	1.0
GL_LINE_SMOOTH	Line antialiasing on	line/enable	GL_FALSE
GL_LINE_STIPPLE_PATTERN	Line stipple	line	1's
GL_LINE_STIPPLE_REPEAT	Line stipple repeat	line	1
GL_LINE_STIPPLE	Line stipple enable	line/enable	GL_FALSE
GL_CULL_FACE	Polygon culling enabled	polygon/enable	GL_FALSE
GL_CULL_FACE_MODE	Cull front-/back-facing polygons	polygon	GL_BACK
GL_FRONT_FACE	Polygon front-face CW/CCW indicator	polygon	GL_CCW
GL_POLYGON_SMOOTH	Polygon antialiasing on	polygon/enable	GL_FALSE
GL_POLYGON_MODE	Polygon rasterization mode (front and back)	polygon	GL_FILL
GL_POLYGON_OFFSET_FACTOR	Polygon offset factor	polygon	0
GL_POLYGON_OFFSET_BIAS	Polygon offset bias	polygon	0
			JLJL

GL_POLYGON_OFFSET_POINT	Polygon offset enable for GL_POINT mode rasterization	polygon/enable	GL_FALSE
GL_POLYGON_OFFSET_LINE	Polygon offset enable for GL_LINE mode rasterization	polygon/enable	GL_FALSE
GL_POLYGON_OFFSET_FILL	Polygon offset enable for GL_FILL mode rasterization	polygon/enable	GL_FALSE
GL_POLYGON_STIPPLE	Polygon stipple enable	polygon/enable	GL_FALSE
-	Polygon stipple pattern	polygon-stipple	1's

Texturing

Table B-7 : (continued) Texturing State Variables

State Variable	Description	Attribute Group	Initial Value	G
GL_TEXTURE_x	True if <i>x</i> -D texturing enabled (<i>x</i> is 1D or 2D)	texture/e nable	GL_FALSE	gll
GL_TEXTURE_BINDING_x	Texture object bound to GL_TEXTURE_x (x is 1D or 2D)	texture	GL_FALSE	gl(
GL_TEXTURE	<i>x</i> -D texture image at level of detail <i>i</i>	-	-	gl(
GL_TEXTURE_WIDTH	<i>x</i> -D texture image <i>i</i> 's width	-	0	gl(
GL_TEXTURE_HEIGHT	<i>x</i> -D texture image <i>i</i> 's height	_	0	gl(

GL_TEXTURE_BORDER	<i>x</i> -D texture image <i>i</i> 's border width	-	0	gl(
GL_TEXTURE_INTERNAL _FORMAT	<i>x</i> -D texture image <i>i</i> 's internal image format	-	1	g](
GL_TEXTURE_RED_SIZE	<i>x</i> -D texture image <i>i</i> 's red resolution	-	0	gl(
GL_TEXTURE_GREEN_SIZE	<i>x</i> -D texture image <i>i</i> 's green resolution	-	0	gl(
GL_TEXTURE_BLUE_SIZE	<i>x</i> -D texture image <i>i</i> 's blue resolution	-	0	gl(
GL_TEXTURE_ALPHA_SIZE	<i>x</i> -D texture image <i>i</i> 's alpha resolution	-	0	gl(
GL_TEXTURE_LUMINANCE_SIZE	<i>x</i> -D texture image <i>i</i> 's luminance resolution	-	0	gl(
GL_TEXTURE_INTENSITY_SIZE	<i>x</i> -D texture image <i>i</i> 's intensity resolution	-	0	gl(
GL_TEXTURE_BORDER_COLOR	Texture border color	texture	0, 0, 0, 0	gl(
GL_TEXTURE_MIN_FILTER	Texture minification function	texture	GL_ NEAREST_ MIPMAP_ LINEAR	gl(
GL_TEXTURE_MAG_FILTER	Texture magnification function	texture	GL_LINEAR	gl(
GL_TEXTURE_WRAP_x	Texture wrap mode (x is S or T)	texture	GL_REPEAT	gl(
GL_TEXTURE_PRIORITY	Texture object priority	texture	1	gl(

GL_TEXTURE_RESIDENCY	Texture residency	texture	GL_FALSE	gl(
GL_TEXTURE_ENV_MODE	Texture application function	texture	GL_ MODULATE	gl(
GL_TEXTURE_ENV_COLOR	Texture environment color	texture	0, 0, 0, 0	gl(
GL_TEXTURE_GEN_x	Texgen enabled (x is S, T, R, or Q)	texture/e nable	GL_FALSE	gll
GL_EYE_PLANE	Texgen plane equation coefficients	texture	-	gl(
GL_OBJECT_PLANE	Texgen object linear coefficients	texture	-	gl(
GL_TEXTURE_GEN_MODE	Function used for texgen	texture	GL_EYE_ LINEAR	gl(

Pixel Operations

s
5

State Variable	Description	Attribute Group	Initial Value	G
GL_SCISSOR_TEST	Scissoring enabled	scissor/enable	GL_FALSE	g
GL_SCISSOR_BOX	Scissor box	scissor	-	g
GL_ALPHA_TEST	Alpha test enabled	color-buffer/ enable	GL_FALSE	g
GL_ALPHA_TEST_FUNC	Alpha test function	color-buffer	GL_ALWAYS	g
GL_ALPHA_TEST_REF	Alpha test reference value	color-buffer	0	g

GL_STENCIL_TEST	Stenciling enabled	stencil-buffer/en able	GL_FALSE	g
GL_STENCIL_FUNC	Stencil function	stencil-buffer	GL_ALWAYS	g
GL_STENCIL_VALUE_MASK	Stencil mask	stencil-buffer	1's	g
GL_STENCIL_REF	Stencil reference value	stencil-buffer	0	g
GL_STENCIL_FAIL	Stencil fail action	stencil-buffer	GL_KEEP	g
GL_STENCIL_PASS_DEPTH_FAIL	Stencil depth buffer fail action	stencil-buffer	GL_KEEP	g
GL_STENCIL_PASS_DEPTH_PASS	Stencil depth buffer pass action	stencil-buffer	GL_KEEP	g
GL_DEPTH_TEST	Depth buffer enabled	depth-buffer/ena ble	GL_FALSE	g
GL_DEPTH_FUNC	Depth buffer test function	depth-buffer	GL_LESS	g
GL_BLEND	Blending enabled	color-buffer/ enable	GL_FALSE	g
GL_BLEND_SRC	Blending source function	color-buffer	GL_ONE	g
GL_BLEND_DST	Blending destination function	color-buffer	GL_ZERO	g
GL_DITHER	Dithering enabled	color-buffer/ enable	GL_TRUE	g

GL_INDEX_LOGIC_OP	Color index logical operation enabled	color-buffer/ enable	GL_FALSE	g
GL_COLOR_LOGIC_OP	RGBA color logical operation enabled	color-buffer/ enable	GL_FALSE	g
GL_LOGIC_OP_MODE	Logical operation function	color-buffer	GL_COPY	g

Framebuffer Control

Table B-9: H	Framebuffer	Control State	Variables
--------------	-------------	---------------	-----------

State Variable	Description	Attribute Group	Initial Value	Get
GL_DRAW_BUFFER	Buffers selected for drawing	color-buffer	-	glGe
GL_INDEX_WRITEMASK	Color-index writemask	color-buffer	1's	glGe
GL_COLOR_WRITEMASK	Color write enables; R, G, B, or A	color-buffer	GL_TRUE	glGe
GL_DEPTH_WRITEMASK	Depth buffer enabled for writing	depth-buffer	GL_TRUE	glGe
GL_STENCIL_WRITEMASK	Stencil-buffer writemask	stencil-buffer	1's	glGe
GL_COLOR_CLEAR_VALUE	Color-buffer clear value (RGBA mode)	color-buffer	0, 0, 0, 0	glGe
GL_INDEX_CLEAR_VALUE	Color-buffer clear value (color-index mode)	color-buffer	0	glGe
GL_DEPTH_CLEAR_VALUE	Depth-buffer clear value	depth-buffer	1	glGe
GL_STENCIL_CLEAR_VALUE	Stencil-buffer clear value	stencil-buffer	0	glGe
GL_ACCUM_CLEAR_VALUE	Accumulation-buffer clear value	accum-buffer	0	glGe

Pixels

Table B-10 : (continued) Pixel State Variables

State Variable	Description	Attribute Group	Initial Value
GL_UNPACK_SWAP_BYTES	Value of GL_UNPACK_SWAP_BYTES	pixel-store	GL_FALS

GL_UNPACK_LSB_FIRST	Value of GL_UNPACK_LSB_FIRST	pixel-store	GL_FALS
GL_UNPACK_ROW_LENGTH	Value of GL_UNPACK_ROW_LENGTH	pixel-store	0
GL_UNPACK_SKIP_ROWS	Value of GL_UNPACK_SKIP_ROWS	pixel-store	0
GL_UNPACK_SKIP_PIXELS	Value of GL_UNPACK_SKIP_PIXELS	pixel-store	0
GL_UNPACK_ALIGNMENT	Value of GL_UNPACK_ALIGNMENT	pixel-store	4
GL_PACK_SWAP_BYTES	Value of GL_PACK_SWAP_BYTES	pixel-store	GL_FALS
GL_PACK_LSB_FIRST	Value of GL_PACK_LSB_FIRST	pixel-store	GL_FALS
GL_PACK_ROW_LENGTH	Value of GL_PACK_ROW_LENGTH	pixel-store	0
GL_PACK_SKIP_ROWS	Value of GL_PACK_SKIP_ROWS	pixel-store	0
GL_PACK_SKIP_PIXELS	Value of GL_PACK_SKIP_PIXELS	pixel-store	0
GL_PACK_ALIGNMENT	Value of GL_PACK_ALIGNMENT	pixel-store	4
GL_MAP_COLOR	True if colors are mapped	pixel	GL_FALS
GL_MAP_STENCIL	True if stencil values are mapped	pixel	GL_FALS
GL_INDEX_SHIFT	Value of GL_INDEX_SHIFT	pixel	0
GL_INDEX_OFFSET	Value of GL_INDEX_OFFSET	pixel	0
GL_x_SCALE	Value of GL_x_SCALE; x is GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_DEPTH	pixel	1

GL_x_BIAS	Value of GL_x_BIAS; x is one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, or GL_DEPTH	pixel	0
GL_ZOOM_X	x zoom factor	pixel	1.0
GL_ZOOM_Y	y zoom factor	pixel	1.0
GL_x	glPixelMap() translation tables; x is a map name from Table 8-1	-	0's
GL_x_SIZE	Size of table <i>x</i>	-	1
GL_READ_BUFFER	Read source buffer	pixel	-

Evaluators

State Variable	Description	Attribute Group	Initial Value	Get Command
GL_ORDER	1D map order	-	1	glGetMapiv()
GL_ORDER	2D map orders	-	1, 1	glGetMapiv()
GL_COEFF	1D control points	-	-	glGetMapfv()
GL_COEFF	2D control points	-	-	glGetMapfv()
GL_DOMAIN	1D domain endpoints	-	-	glGetMapfv()
GL_DOMAIN	2D domain endpoints	-	-	glGetMapfv()

 Table B-11 : Evaluator State Variables

GL_MAP1_x	1D map enables: <i>x</i> is map type	eval/enable	GL_FALSE	glIsEnabled()
GL_MAP2_x	2D map enables: <i>x</i> is map type	eval/enable	GL_FALSE	glIsEnabled()
GL_MAP1_GRID_DOMAIN	1D grid endpoints	eval	0, 1	glGetFloatv()
GL_MAP2_GRID_DOMAIN	2D grid endpoints	eval	0, 1; 0, 1	glGetFloatv()
GL_MAP1_GRID_SEGMENTS	1D grid divisions	eval	1	glGetFloatv()
GL_MAP2_GRID_SEGMENTS	2D grid divisions	eval	1,1	glGetFloatv()
GL_AUTO_NORMAL	True if automatic normal generation enabled	eval	GL_FALSE	glIsEnabled()

Hints

Table B-12 : Hint State Variables

State Variable	Description	Attribute Group	Initial Value
GL_PERSPECTIVE_CORRECTION_HINT	Perspective correction hint	hint	GL_DONT_CARE
GL_POINT_SMOOTH_HINT	Point smooth hint	hint	GL_DONT_CARE
GL_LINE_SMOOTH_HINT	Line smooth hint	hint	GL_DONT_CARE
GL_POLYGON_SMOOTH_HINT	Polygon smooth hint	hint	GL_DONT_CARE
GL_FOG_HINT	Fog hint	hint	GL_DONT_CARE

Implementation-Dependent Values

Table B-13 :	(continued)	Implementation-Depe	endent State Variables
--------------	-------------	---------------------	------------------------

State Variable	Description	Attribute Group	Minimuı Value
GL_MAX_LIGHTS	Maximum number of lights	-	8
GL_MAX_CLIP_PLANES	Maximum number of user clipping planes	-	6
GL_MAX_MODELVIEW_STACK_DEPTH	Maximum modelview-matrix stack depth	-	32
GL_MAX_PROJECTION_STACK_DEPTH	Maximum projection-matrix stack depth	-	2
GL_MAX_TEXTURE_STACK_DEPTH	Maximum depth of texture matrix stack	-	2

		ir	
GL_SUBPIXEL_BITS	Number of bits of subpixel precision in x and y	-	4
GL_MAX_TEXTURE_SIZE	See discussion in "Texture Proxy" in Chapter 9	-	64
GL_MAX_PIXEL_MAP_TABLE	Maximum size of a glPixelMap() translation table	-	32
GL_MAX_NAME_STACK_DEPTH	Maximum selection-name stack depth	-	64
GL_MAX_LIST_NESTING	Maximum display-list call nesting	-	64
GL_MAX_EVAL_ORDER	Maximum evaluator polynomial order	-	8
GL_MAX_VIEWPORT_DIMS	Maximum viewport dimensions	-	-
GL_MAX_ATTRIB_STACK_DEPTH	Maximum depth of the attribute stack	-	16
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	Maximum depth of the client attribute stack	-	16
GL_AUX_BUFFERS	Number of auxiliary buffers	-	0
GL_RGBA_MODE	True if color buffers store RGBA	-	-
GL_INDEX_MODE	True if color buffers store indices	-	-

GL_DOUBLEBUFFER	True if front and back buffers exist	-	-
GL_STEREO	True if left and right buffers exist	-	-
GL_POINT_SIZE_RANGE	Range (low to high) of antialiased point sizes	-	1, 1
GL_POINT_SIZE_GRANULARITY	Antialiased point-size granularity	-	-
GL_LINE_WIDTH_RANGE	Range (low to high) of antialiased line widths	-	1, 1
GL_LINE_WIDTH_GRANULARITY	Antialiased line-width granularity	-	-

Implementation-Dependent Pixel Depths

State Variable	Description	Attribute Group	Minimum Value	Get Command
GL_RED_BITS	Number of bits per red component in color buffers	-	-	glGetIntegerv()
GL_GREEN_BITS	Number of bits per green component in color buffers	-	-	glGetIntegerv()
GL_BLUE_BITS	Number of bits per blue component in color buffers	-	-	glGetIntegerv()

GL_ALPHA_BITS	Number of bits per alpha component in color buffers	-	-	glGetIntegerv()
GL_INDEX_BITS	Number of bits per index in color buffers	-	-	glGetIntegerv()
GL_DEPTH_BITS	Number of depth-buffer bitplanes	-	-	glGetIntegerv()
GL_STENCIL_BITS	Number of stencil bitplanes	-	-	glGetIntegerv()
GL_ACCUM_RED_BITS	Number of bits per red component in the accumulation buffer	-	-	glGetIntegerv()
GL_ACCUM_GREEN_BITS	Number of bits per green component in the accumulation buffer	-	-	glGetIntegerv()
GL_ACCUM_BLUE_BITS	Number of bits per blue component in the accumulation buffer	-	-	glGetIntegerv()
GL_ACCUM_ALPHA_BITS	Number of bits per alpha component in the accumulation buffer	-	-	glGetIntegerv()

Miscellaneous

State Variable	Description	Attribute Group	Initial Value	
GL_LIST_BASE	Setting of glListBase()	list	0	g
GL_LIST_INDEX	Number of display list under construction; 0 if none	-	0	£
GL_LIST_MODE	Mode of display list under construction; undefined if none	-	0	Į
GL_ATTRIB_STACK_DEPTH	Attribute stack pointer	-	0	Ę
GL_CLIENT_ATTRIB_STACK_DEPTH	Client attribute stack pointer	-	0	Ę
GL_NAME_STACK_DEPTH	Name stack depth	-	0	£
GL_RENDER_MODE	glRenderMode() setting	-	GL_RENDER	£
GL_SELECTION_BUFFER_POINTER	Pointer to selection buffer	select	0	Ę
GL_SELECTION_BUFFER_SIZE	Size of selection buffer	select	0	Į
GL_FEEDBACK_BUFFER_POINTER	Pointer to feedback buffer	feedback	0	£
GL_FEEDBACK_BUFFER_SIZE	Size of feedback buffer	feedback	0	Į
GL_FEEDBACK_BUFFER_TYPE	Type of feedback buffer	feedback	GL_2D	Į

-	Current error code(s)	-	0	٤
---	-----------------------	---	---	---

← →	OpenGL Programming Guide
	(Addison-Wesley Publishing Company)

Appendix C OpenGL and Window Systems

OpenGL is available on many different platforms and works with many different window systems. OpenGL is designed to complement window systems, not duplicate their functionality. Therefore, OpenGL performs geometric and image rendering in two and three dimensions, but it does not manage windows or handle input events.

However, the basic definitions of most window systems don't support a library as sophisticated as OpenGL, with its complex and diverse pixel formats, including depth, stencil, and accumulation buffers, as well as double-buffering. For most window systems, some routines are added to extend the window system to support OpenGL.

This appendix introduces the extensions defined for several window and operating systems: the X Window System, the Apple Mac OS, OS/2 Warp from IBM, and Microsoft Windows NT and Windows 95. You need to have some knowledge of the window systems to fully understand this appendix.

This appendix has the following major sections:

- "GLX: OpenGL Extension for the X Window System"
- "AGL: OpenGL Extension to the Apple Macintosh"
- "PGL: OpenGL Extension for IBM OS/2 Warp"
- "WGL: OpenGL Extension for Microsoft Windows NT and Windows 95"

GLX: OpenGL Extension for the X Window System

In the X Window System, OpenGL rendering is made available as an extension to X in the formal X sense. GLX is an extension to the X protocol (and its associated API) for communicating OpenGL commands to an extended X server. Connection and authentication are accomplished with the normal X mechanisms.

As with other X extensions, there is a defined network protocol for OpenGL's rendering commands encapsulated within the X byte stream, so client-server OpenGL rendering is supported. Since performance is critical in three-dimensional rendering, the OpenGL extension to X allows OpenGL to bypass the X server's involvement in data encoding, copying, and interpretation and instead render directly to the graphics pipeline. The X Visual is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type XVisualInfo keeps track of pixel information, including pixel type (RGBA or color index), single or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers. The standard X Visuals (for example, PseudoColor, TrueColor) do not describe the pixel format details, so each implementation must extend the number of X Visuals supported.

The GLX routines are discussed in more detail in the *OpenGL Reference Manual*. Integrating OpenGL applications with the X Window System and the Motif widget set is discussed in great detail in *OpenGL Programming for the X Window System* by Mark Kilgard (Reading, MA: Addison-Wesley Developers Press, 1996), which includes full source code examples. If you absolutely want to learn about the internals of GLX, you may want to read the GLX specification, which can be found at

ftp://sgigate.sgi.com/pub/opengl/doc/

Initialization

Use glXQueryExtension() and glXQueryVersion() to determine whether the GLX extension is defined for an X server and, if so, which version is present. glXQueryExtensionsString() returns extension information about the client-server connection. glXGetClientString() returns information about the client library, including extensions and version number. glXQueryServerString() returns similar information about the server.

glXChooseVisual() returns a pointer to an XVisualInfo structure describing the visual that meets the client's specified attributes. You can query a visual about its support of a particular OpenGL attribute with **glXGetConfig**().

Controlling Rendering

Several GLX routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render off-screen if you want. Routines are also provided for such tasks as synchronizing execution between the X and OpenGL streams, swapping front and back buffers, and using an X font.

Managing an OpenGL Rendering Context

An OpenGL rendering context is created with **glXCreateContext**(). One of the arguments to this routine allows you to request a direct rendering context that bypasses the X server as described previously. (Note that to do direct rendering, the X server connection must be local, and the OpenGL implementation needs to support direct rendering.) **glXCreateContext**() also allows display-list and texture-object indices and definitions to be shared by multiple rendering contexts. You can determine whether a GLX context is direct with **glXIsDirect**().

To make a rendering context current, use **glXMakeCurrent()**; **glXGetCurrentContext()** returns the current context. You can also obtain the current drawable with **glXGetCurrentDrawable()** and the current X Display with **glXGetCurrentDisplay()**. Remember that only one context can be current for any thread at any one time. If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with **glXCopyContext()**. When you're finished with a particular context, destroy it with **glXDestroyContext()**.

Off-Screen Rendering

To render off-screen, first create an X Pixmap and then pass this as an argument to **glXCreateGLXPixmap()**. Once rendering is completed, you can destroy the association between the X and GLX Pixmaps with **glXDestroyGLXPixmap()**. (Off-screen rendering isn't guaranteed to be supported for direct renderers.)

Synchronizing Execution

To prevent X requests from executing until any outstanding OpenGL rendering is completed, call **glXWaitGL()**. Then, any previously issued OpenGL commands are guaranteed to be executed before any X rendering calls made after **glXWaitGL()**. Although the same result can be achieved with **glFinish()**, **glXWaitGL()** doesn't require a round trip to the server and thus is more efficient in cases where the client and server are on separate machines.

To prevent an OpenGL command sequence from executing until any outstanding X requests are completed, use **glXWaitX()**. This routine guarantees that previously issued X rendering calls are executed before any OpenGL calls made after **glXWaitX()**.

Swapping Buffers

For drawables that are double-buffered, the front and back buffers can be exchanged by calling **glXSwapBuffers()**. An implicit **glFlush()** is done as part of this routine.

Using an X Font

A shortcut for using X fonts in OpenGL is provided with the command **glXUseXFont**(). This routine builds display lists, each of which calls **glBitmap**(), for each requested character from the specified font and font size.

GLX Prototypes

Initialization

Determine whether the GLX extension is defined on the X server:

Bool **glXQueryExtension** (Display **dpy*, int **errorBase*, int **eventBase*);

Query version and extension information for client and server:

Bool **glXQueryVersion** (Display **dpy*, int **major*, int **minor*);

const char* glXGetClientString (Display *dpy, int name);

const char* glXQueryServerString (Display *dpy, int screen, int name);

const char* glXQueryExtensionsString (Display *dpy, int screen);

Obtain the desired visual:

XVisualInfo* **glXChooseVisual** (Display **dpy*, int *screen*, int **attribList*);

int glXGetConfig (Display *dpy, XVisualInfo *visual, int attrib, int *value);

Controlling Rendering

Manage or query an OpenGL rendering context:

GLXContext **glXCreateContext** (Display **dpy*, XVisualInfo **visual*, GLXContext *shareList*, Bool *direct*);

void glXDestroyContext (Display *dpy, GLXContext context);

void glXCopyContext (Display *dpy, GLXContext source, GLXContext dest, unsigned long mask);

Bool **glXIsDirect** (Display **dpy*, GLXContext *context*);

Bool **glXMakeCurrent** (Display **dpy*, GLXDrawable *draw*, GLXContext *context*);

GLXContext glXGetCurrentContext (void);

Display* glXGetCurrentDisplay (void);

GLXDrawable glXGetCurrentDrawable (void);

Perform off-screen rendering:

GLXPixmap **glXCreateGLXPixmap** (Display **dpy*, XVisualInfo **visual*, Pixmap *pixmap*);

void glXDestroyGLXPixmap (Display *dpy, GLXPixmap pix);

Synchronize execution:

void glXWaitGL (void);

void glXWaitX (void);

Exchange front and back buffers:

void **glXSwapBuffers** (Display **dpy*, GLXDrawable *drawable*);

Use an X font:

void glXUseXFont (Font font, int first, int count, int listBase);

AGL: OpenGL Extension to the Apple Macintosh

This section covers the routines defined as the OpenGL extension to the Apple Macintosh (AGL), as defined by Template Graphics Software. An understanding of the way the Macintosh handles graphics rendering (QuickDraw) is required. The *Macintosh Toolbox Essentials* and *Imaging With QuickDraw* manuals from the *Inside Macintosh* series are also useful to have at hand.

For more information (including how to obtain the OpenGL software library for the Power Macintosh), you may want to check out the web site for OpenGL information at Template Graphics Software:

http://www.sd.tgs.com/Products/opengl.htm

For the Macintosh, OpenGL rendering is made available as a library that is either compiled in or resident as an extension for an application that wishes to make use of it. OpenGL is implemented in software for systems that do not possess hardware acceleration. Where acceleration is available (through the QuickDraw 3D Accelerator), those capabilities that match the OpenGL pipeline are used with the remaining functionality being provided through software rendering.

The data type AGLPixelFmtID (the AGL equivalent to XVisualInfo) maintains pixel information, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

In contrast to other OpenGL implementations on other systems (such as the X Window System), the client/server model is not used. However, you may still need to call **glFlush**() since some hardware accelerators buffer the OpenGL pipeline and require a flush to empty it.

Initialization

Use aglQueryVersion() to determine what version of OpenGL for the Macintosh is available.

The capabilities of underlying graphics devices and your requirements for rendering buffers are resolved using **aglChoosePixelFmt(**). Use **aglListPixelFmts(**) to find the particular formats supported by a graphics device. Given a pixel format, you can determine which attributes are available by using **aglGetConfig(**).

Rendering and Contexts

Several AGL routines are provided for creating and managing an OpenGL rendering context. You can use such a context to render into either a window or an off-screen graphics world. Routines are also provided that allow you to swap front and back rendering buffers, adjust buffers in response to a move, resize or graphics device change event, and use Macintosh fonts. For software rendering (and in some cases, hardware-accelerated rendering) the rendering buffers are created in your application memory space. For the application to work properly you must provide sufficient memory for these buffers in your application's SIZE resource.

Managing an OpenGL Rendering Context

An OpenGL rendering context is created (at least one context per window being rendered into) with **aglCreateContext**(). This takes the pixel format you selected as a parameter and uses it to initialize the context.

Use **aglMakeCurrent**() to make a rendering context current. Only one context can be current for a thread of control at any time. This indicates which drawable is to be rendered into and which context to use with it. It's possible for more than one context to be used (not simultaneously) with a particular drawable. Two routines allow you to determine which is the current rendering context and drawable being rendered into: **aglGetCurrentContext**() and **aglGetCurrentDrawable**().

If you have multiple contexts, you can copy selected groups of OpenGL state variables from one context to another with **aglCopyContext**(). When a particular context is finished with, it should be destroyed by calling **aglDestroyContext**().

On-screen Rendering

With the OpenGL extensions for the Apple Macintosh you can choose whether window clipping is performed when writing to the screen and whether the cursor is hidden during screen writing operations. This is important since these two items may affect how fast rendering can be performed. Call **aglSetOptions**() to select these options.

Off-screen Rendering

To render off-screen, first create an off-screen graphics world in the usual way, and pass the handle into **aglCreateAGLPixmap()**. This routine returns a drawable that can be used with **aglMakeCurrent()**. Once rendering is completed, you can destroy the association with **aglDestroyAGLPixmap()**.

Swapping Buffers

For drawables that are double-buffered (as per the pixel format of the current rendering context), call **aglSwapBuffers**() to exchange the front and back buffers. An implicit **glFlush**() is performed as part of this routine.

Updating the Rendering Buffers

The Apple Macintosh toolbox requires you to perform your own event handling and does not provide a way for libraries to automatically hook in to the event stream. So that the drawables maintained by OpenGL can adjust to changes in drawable size, position and pixel depth, **aglUpdateCurrent()** is provided.

This routine must be called by your event processing code whenever one of these events occurs in the current drawable. Ideally the scene should be rerendered after a update call to take into account the changes made to the rendering buffers.

Using an Apple Macintosh Font

A shortcut for using Macintosh fonts is provided with **aglUseFont**(). This routine builds display lists, each of which calls **glBitmap**(), for each requested character from the specified font and font size.

Error Handling

An error-handling mechanism is provided for the Apple Macintosh OpenGL extension. When an

error occurs you can call **aglGetError**() to get a more precise description of what caused the error.

AGL Prototypes

Initialization

Determine AGL version:

GLboolean **aglQueryVersion** (int **major*, int **minor*);

Pixel format selection, availability, and capability:

AGLPixelFmtID **aglChoosePixelFmt** (GDHandle **dev*, int *ndev*, int **attribs*);

int **aglListPixelFmts** (GDHandle *dev*, AGLPixelFmtID ** *fmts*);

GLboolean **aglGetConfig** (AGLPixelFmtID *pix*, int *attrib*, int **value*);

Controlling Rendering

Manage an OpenGL rendering context:

AGLContext **aglCreateContext** (AGLPixelFmtID *pix*, AGLContext *shareList*);

GLboolean **aglDestroyContext** (AGLContext *context*);

GLboolean **aglCopyContext** (AGLContext *source*, AGLContext *dest*, GLuint *mask*);

GLboolean **aglMakeCurrent** (AGLDrawable *drawable*, AGLContext *context*);

GLboolean aglSetOptions (int opts);

AGLContext aglGetCurrentContext (void);

AGLDrawable aglGetCurrentDrawable (void);

Perform off-screen rendering:

AGLPixmap **aglCreateAGLPixmap** (AGLPixelFmtID *pix*, GWorldPtr *pixmap*);

GLboolean **aglDestroyAGLPixmap** (AGLPixmap *pix*);

Exchange front and back buffers:

GLboolean **aglSwapBuffers** (AGLDrawable *drawable*);

Update the current rendering buffers:

GLboolean aglUpdateCurrent (void);

Use a Macintosh font:

GLboolean **aglUseFont** (int *familyID*, int *size*, int *first*, int *count*, int *listBase*);

Find the cause of an error:

GLenum **aglGetError** (void);

PGL: OpenGL Extension for IBM OS/2 Warp

OpenGL rendering for IBM OS/2 Warp is accomplished by using PGL routines added to integrate OpenGL into the standard IBM Presentation Manager. OpenGL with PGL supports both a direct OpenGL context (which is often faster) and an indirect context (which allows some integration of Gpi and OpenGL rendering).

The data type VISUALCONFIG (the PGL equivalent to XVisualInfo) maintains the visual configuration, including pixel type (RGBA or color index), single- or double-buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

To get more information (including how to obtain the OpenGL software library for IBM OS/2 Warp, Version 3.0), you may want to start at

http://www.austin.ibm.com/software/OpenGL/

Packaged along with the software is the document, *OpenGL On OS/2 Warp*, which provides more detailed information. OpenGL support is included with the base operating system with OS/2 Warp Version 4.

Initialization

Use **pglQueryCapability()** and **pglQueryVersion()** to determine whether the OpenGL is supported on this machine and, if so, how it is supported and which version is present. **pglChooseConfig()** returns a pointer to an VISUALCONFIG structure describing the visual configuration that best meets the client's specified attributes. A list of the particular visual configurations supported by a graphics device can be found using **pglQueryConfigs()**.

Controlling Rendering

Several PGL routines are provided for creating and managing an OpenGL rendering context, capturing the contents of a bitmap, synchronizing execution between the Presentation Manager and OpenGL streams, swapping front and back buffers, using a color palette, and using an OS/2 logical font.

Managing an OpenGL Rendering Context

An OpenGL rendering context is created with **pglCreateContext()**. One of the arguments to this routine allows you to request a direct rendering context that bypasses the Gpi and render to a PM window, which is generally faster. You can determine whether a OpenGL context is direct with **pglIsIndirect()**.

To make a rendering context current, use **pglMakeCurrent()**; **pglGetCurrentContext()** returns the current context. You can also obtain the current window with **pglGetCurrentWindow()**. You can copy some OpenGL state variables from one context to another with **pglCopyContext()**. When you're finished with a particular context, destroy it with **pglDestroyContext()**.

Access the Bitmap of the Front Buffer

To lock access to the bitmap representation of the contents of the front buffer, use **pglGrabFrontBitmap(**). An implicit **glFlush(**) is performed, and you can read the bitmap, but its contents are effectively read-only. Immediately after access is completed, you should call **pglReleaseFrontBitmap(**) to restore write access to the front buffer.

Synchronizing Execution

To prevent Gpi rendering requests from executing until any outstanding OpenGL rendering is completed, call **pglWaitGL**(). Then, any previously issued OpenGL commands are guaranteed to be executed before any Gpi rendering calls made after **pglWaitGL**().

To prevent an OpenGL command sequence from executing until any outstanding Gpi requests are completed, use **pglWaitPM**(). This routine guarantees that previously issued Gpi rendering calls are executed before any OpenGL calls made after **pglWaitPM**().

Note: OpenGL and Gpi rendering can be integrated in the same window only if the OpenGL context is an indirect context.

Swapping Buffers

For windows that are double-buffered, the front and back buffers can be exchanged by calling **pglSwapBuffers**(). An implicit **glFlush**() is done as part of this routine.

Using a Color Index Palette

When you are running in 8-bit (256 color) mode, you have to worry about color palette management. For windows with a color index Visual Configuration, call pglSelectColorIndexPalette() to tell OpenGL what color-index palette you want to use with your context. A color palette must be selected before the context is initially bound to a window. In RGBA mode, OpenGL sets up a palette automatically.

Using an OS/2 Logical Font

A shortcut for using OS/2 logical fonts in OpenGL is provided with the command **pglUseFont**(). This routine builds display lists, each of which calls **glBitmap**(), for each requested character from the specified font and font size.

PGL Prototypes

Initialization

Determine whether OpenGL is supported and, if so, its version number:

long **pglQueryCapability** (HAB *hab*);

void **pglQueryVersion** (HAB *hab*, int **major*, int **minor*);

Visual configuration selection, availability and capability:

PVISUALCONFIG **pglChooseConfig** (HAB *hab*, int **attribList*);

PVISUALCONFIG * pglQueryConfigs (HAB hab);

Controlling Rendering

Manage or query an OpenGL rendering context:

HGC **pglCreateContext** (HAB *hab*, PVISUALCONFIG *pVisualConfig*, HGC *shareList*, Bool isDirect);

Bool **pglDestroyContext** (HAB *hab*, HGC *hgc*);

Bool pglCopyContext (HAB hab, HGC source, HGC dest, GLuint mask);

Bool pglMakeCurrent (HAB hab, HGC hgc, HWND hwnd);

long **pglIsIndirect** (HAB *hab*, HGC *hgc*);

HGC pglGetCurrentContext (HAB hab);

HWND pglGetCurrentWindow (HAB hab);

Access and release the bitmap of the front buffer:

Bool **pglGrabFrontBitmap** (HAB *hab*, HPS **hps*, HBITMAP **phbitmap*);

Bool **pglReleaseFrontBitmap** (HAB *hab*);

Synchronize execution:

HPS pglWaitGL (HAB hab);

void **pglWaitPM** (HAB *hab*);

Exchange front and back buffers:

void **pglSwapBuffers** (HAB hab, HWND hwnd);

Finding a color-index palette:

Use an OS/2 logical font:

Bool **pglUseFont** (HAB *hab*, HPS *hps*, FATTRS **fontAttribs*, long *logicalId*, int *first*, int *count*, int *listBase*);

WGL: OpenGL Extension for Microsoft Windows NT and Windows 95

OpenGL rendering is supported on systems that run Microsoft Windows NT and Windows 95. The functions and routines of the Win32 library are necessary to initialize the pixel format and control rendering for OpenGL. Some routines, which are prefixed by **wgl**, extend Win32 so that OpenGL can be fully supported.

For Win32/WGL, the PIXELFORMATDESCRIPTOR is the key data structure to maintain pixel format information about the OpenGL window. A variable of data type PIXELFORMATDESCRIPTOR keeps track of pixel information, including pixel type (RGBA or color index), single- or double- buffering, resolution of colors, and presence of depth, stencil, and accumulation buffers.

To get more information about WGL, you may want to start with technical articles available through the Microsoft Developer Network at

http://www.microsoft.com/msdn/

Initialization

Use **GetVersion**() or the newer **GetVersionEx**() to determine version information. **ChoosePixelFormat**() tries to find a PIXELFORMATDESCRIPTOR with specified attributes. If a good match for the requested pixel format is found, then **SetPixelFormat**() should be called to actually use the pixel format. You should select a pixel format in the device context before calling **wglCreateContext**().

If you want to find out details about a given pixel format, use **DescribePixelFormat()** or, for overlays or underlays, **wglDescribeLayerPlane()**.

Controlling Rendering

Several WGL routines are provided for creating and managing an OpenGL rendering context, rendering to a bitmap, swapping front and back buffers, finding a color palette, and using either bitmap or outline fonts.

Managing an OpenGL Rendering Context

wglCreateContext() creates an OpenGL rendering context for drawing on the device in the selected pixel format of the device context. (To create an OpenGL rendering context for overlay or underlay windows, use **wglCreateLayerContext()** instead.) To make a rendering context current,

use **wglMakeCurrent()**; **wglGetCurrentContext()** returns the current context. You can also obtain the current device context with **wglGetCurrentDC()**. You can copy some OpenGL state variables from one context to another with **wglCopyContext()** or make two contexts share the same display lists and texture objects with **wglShareLists()**. When you're finished with a particular context, destroy it with **wglDestroyContext()**.

OpenGL Rendering to a Bitmap

Win32 has a few routines to allocate (and deallocate) bitmaps, to which you can render OpenGL directly. **CreateDIBitmap()** creates a device-dependent bitmap (DDB) from a device-independent bitmap (DIB). **CreateDIBSection()** creates a device-independent bitmap (DIB) that applications can write to directly. When finished with your bitmap, you can use **DeleteObject()** to free it up.

Synchronizing Execution

If you want to combine GDI and OpenGL rendering, be aware there are no equivalents to functions like **glXWaitGL()**, **glXWaitX()**, or **pglWaitGL()** in Win32. Although **glXWaitGL()** has no equivalent in Win32, you can achieve the same effect by calling **glFinish()**, which waits until all pending OpenGL commands are executed, or by calling **GdiFlush()**, which waits until all GDI drawing has completed.

Swapping Buffers

For windows that are double-buffered, the front and back buffers can be exchanged by calling **SwapBuffers(**) or **wglSwapLayerBuffers(**); the latter for overlays and underlays.

Finding a Color Palette

To access the color palette for the standard (non-layer) bitplanes, use the standard GDI functions to set the palette entries. For overlay or underlay layers, use **wglRealizeLayerPalette**(), which maps palette entries from a given color-index layer plane into the physical palette or initializes the palette of an RGBA layer plane. **wglGetLayerPaletteEntries**() is used to query the entries in palettes of layer planes.

Using a Bitmap or Outline Font

WGL has two routines, **wglUseFontBitmaps**() and **wglUseFontOutlines**(), for converting system fonts to use with OpenGL. Both routines build a display list for each requested character from the specified font and font size.

WGL Prototypes

Initialization

Determine version information:

BOOL GetVersion (LPOSVERSIONINFO lpVersionInformation);

BOOL GetVersionEx (LPOSVERSIONINFO *lpVersionInformation*);

Pixel format availability, selection, and capability:

int ChoosePixelFormat (HDC hdc, CONST PIXELFORMATDESCRIPTOR * ppfd);

BOOL **SetPixelFormat** (HDC *hdc*, int *iPixelFormat*, CONST PIXELFORMATDESCRIPTOR * *ppfd*);

int DescribePixelFormat (HDC hdc, int iPixelFormat, UINT nBytes, LPPIXELFORMATDESCRIPTOR ppfd);

BOOL **wglDescribeLayerPlane** (HDC *hdc*, int *iPixelFormat*, int *iLayerPlane*, UINT *nBytes*, LPLAYERPLANEDESCRIPTOR *plpd*);

Controlling Rendering

Manage or query an OpenGL rendering context:

HGLRC wglCreateContext (HDC hdc);

HGLRC wglCreateLayerContext (HDC hdc, int iLayerPlane);

BOOL wglShareLists (HGLRC *hglrc1*, HGLRC *hglrc2*);

BOOL wglDeleteContext (HGLRC *hglrc*);

BOOL **wglCopyContext** (HGLRC *hglrcSource*, HGLRC *hlglrcDest*, UINT *mask*);

BOOL wglMakeCurrent (HDC hdc, HGLRC hglrc);

HGLRC wglGetCurrentContext (VOID);

HDC wglGetCurrentDC (VOID);

Access and release the bitmap of the front buffer:

HBITMAP CreateDIBitmap (HDC hdc, CONST BITMAPINFOHEADER *lpbmih, DWORD fdwInit, CONST VOID *lpbInit, CONST BITMAPINFO *lpbmi, UINT fuUsage);

HBITMAP **CreateDIBSection** (HDC *hdc*, CONST BITMAPINFO **pbmi*, UINT *iUsage*, VOID **ppvBits*, HANDLE *hSection*, DWORD *dwOffset*);

BOOL **DeleteObject** (HGDIOBJ *hObject*);

Exchange front and back buffers:

BOOL **SwapBuffers** (HDC *hdc*);

BOOL wglSwapLayerBuffers (HDC hdc, UINT fuPlanes);

Finding a color palette for overlay or underlay layers:

int **wglGetLayerPaletteEntries** (HDC *hdc*, int *iLayerPlane*, int *iStart*, int *cEntries*, CONST COLORREF **pcr*);

BOOL **wglRealizeLayerPalette** (HDC *hdc*, int *iLayerPlane*, BOOL *bRealize*);

Use a bitmap or an outline font:

BOOL **wglUseFontBitmaps** (HDC *hdc*, DWORD *first*, DWORD *count*, DWORD *listBase*);

BOOL **wglUseFontOutlines** (HDC *hdc*, DWORD *first*, DWORD *count*, DWORD *listBase*, FLOAT *deviation*, FLOAT *extrusion*, int *format*, LPGLYPHMETRICSFLOAT *lpgmf*);

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Appendix D

Basics of GLUT: The OpenGL Utility Toolkit

This appendix describes a subset of Mark Kilgard's OpenGL Utility Toolkit (GLUT), which is fully documented in his book, *OpenGL Programming for the X Window System* (Reading, MA: Addison-Wesley Developers Press, 1996). GLUT has become a popular library for OpenGL programmers, because it standardizes and simplifies window and event management. GLUT has been ported atop a variety of OpenGL implementations, including both the X Window System and Microsoft Windows NT.

This appendix has the following major sections:

- "Initializing and Creating a Window"
- "Handling Window and Input Events"
- "Loading the Color Map"
- "Initializing and Drawing Three-Dimensional Objects"
- "Managing a Background Process"
- "Running the Program"

(See "How to Obtain the Sample Code" in the Preface for information about how to obtain the source code for GLUT.)

With GLUT, your application structures its event handling to use callback functions. (This method is similar to using the Xt Toolkit, also known as the X Intrinsics, with a widget set.) For example, first you open a window and register callback routines for specific events. Then, you create a main loop without an exit. In that loop, if an event occurs, its registered callback functions are executed. Upon completion of the callback functions, flow of control is returned to the main loop.

Initializing and Creating a Window

Before you can open a window, you must specify its characteristics: Should it be single-buffered or double-buffered? Should it store colors as RGBA values or as color indices? Where should it appear on your display? To specify the answers to these questions, call **glutInit(**), **glutInitDisplayMode(**), **glutInitWindowSize(**), and **glutInitWindowPosition(**) before you call **glutCreateWindow(**) to open the window.

void glutInit(int argc, char **argv);

glutInit() should be called before any other GLUT routine, because it initializes the GLUT library. *glutInit()* will also process command line options, but the specific options are window system dependent. For the X Window System, -iconic, -geometry, and -display are examples of command line options, processed by *glutInit()*. (The parameters to the *glutInit()* should be the same as those to *main()*.)

void glutInitDisplayMode(unsigned int mode);

Specifies a display mode (such as RGBA or color-index, or single- or double-buffered) for windows created when **glutCreateWindow**() is called. You can also specify that the window have an associated depth, stencil, and/or accumulation buffer. The mask argument is a bitwise ORed combination of GLUT_RGBA or GLUT_INDEX, GLUT_SINGLE or GLUT_DOUBLE, and any of the buffer-enabling flags: GLUT_DEPTH, GLUT_STENCIL, or GLUT_ACCUM. For example, for a double-buffered, RGBA-mode window with a depth and stencil buffer, use GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL. The default value is GLUT_RGBA | GLUT_SINGLE (an RGBA, single-buffered window). void **glutInitWindowSize**(int width, int height);

void glutInitWindowPosition(int x, int y);

Requests windows created by **glutCreateWindow()** *to have an initial size and position. The arguments* (*x*, *y*) *indicate the location of a corner of the window, relative to the entire display. The width and height indicate the window's size (in pixels). The initial window size and position are hints and may be overridden by other requests.*

int glutCreateWindow(char *name);

Opens a window with previously set characteristics (display mode, width, height, and so on). The string name may appear in the title bar if your window system does that sort of thing. The window is not initially displayed until **glutMainLoop()** is entered, so do not render into the window until then.

The value returned is a unique integer identifier for the window. This identifier can be used for controlling and rendering to multiple windows (each with an OpenGL rendering context) from the same application.

Handling Window and Input Events

After the window is created, but before you enter the main loop, you should register callback functions using the following routines.

void glutDisplayFunc(void (*func)(void));

Specifies the function that's called whenever the contents of the window need to be redrawn. The contents of the window may need to be redrawn when the window is initially opened, when the window is popped and window damage is exposed, and when **glutPostRedisplay()** is explicitly called.

void glutReshapeFunc(void (*func)(int width, int height));

Specifies the function that's called whenever the window is resized or moved. The argument func is a pointer to a function that expects two arguments, the new width and height of the window. Typically, func calls **glViewport(**), so that the display is clipped to the new size, and it redefines the projection matrix so that the aspect ratio of the projected image matches the viewport, avoiding aspect ratio distortion. If **glutReshapeFunc(**) isn't called or is deregistered by passing NULL, a default reshape function is called, which calls **glViewport(**0, 0, width, height).

void glutKeyboardFunc(void (*func)(unsigned int key, int x, int y);

Specifies the function, func, that's called when a key that generates an ASCII character is pressed. The key callback parameter is the generated ASCII value. The x and y callback parameters indicate the location of the mouse (in window-relative coordinates) when the key was pressed.

void **glutMouseFunc**(*void* (*func)(*int* button, *int* state, *int* x, *int* y));

Specifies the function, func, that's called when a mouse button is pressed or released. The button callback parameter is one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. The state callback parameter is either GLUT_UP or GLUT_DOWN, depending upon whether the mouse has been released or pressed. The x and y callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

void glutMotionFunc(void (*func)(int x, int y));

Specifies the function, func, that's called when the mouse pointer moves within the window while one or more mouse buttons is pressed. The x and y callback parameters indicate the location (in window-relative coordinates) of the mouse when the event occurred.

void glutPostRedisplay(void);

Marks the current window as needing to be redrawn. At the next opportunity, the callback function registered by **glutDisplayFunc()** will be called.

Loading the Color Map

If you're using color-index mode, you might be surprised to discover there's no OpenGL routine to load a color into a color lookup table. This is because the process of loading a color map depends entirely on the window system. GLUT provides a generalized routine to load a single color index with an RGB value, **glutSetColor**().

void *glutSetColor*(*GLint index, GLfloat red, GLfloat green, GLfloat blue*); Loads the index in the color map, index, with the given red, green, and blue values. These values are normalized to lie in the range [0.0,1.0].

Initializing and Drawing Three-Dimensional Objects

Many sample programs in this guide use three-dimensional models to illustrate various rendering properties. The following drawing routines are included in GLUT to avoid having to reproduce the code to draw these models in each program. The routines render all their graphics in immediate mode. Each three-dimensional model comes in two flavors: wireframe without surface normals, and solid with shading and surface normals. Use the solid version when you're applying lighting. Only the teapot generates texture coordinates.

void **glutWireSphere**(*GLdouble radius, GLint slices, GLint stacks*); *void* **glutSolidSphere**(*GLdouble radius, GLint slices, GLint stacks*);

void glutWireCube(GLdouble size); void glutSolidCube(GLdouble size); void **glutWireTorus**(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings); void **glutSolidTorus**(GLdouble innerRadius, GLdouble outerRadius, GLint nsides, GLint rings);

void glutWireIcosahedron(void); void glutSolidIcosahedron(void);

void glutWireOctahedron(void); void glutSolidOctahedron(void);

void glutWireTetrahedron(void); void glutSolidTetrahedron(void);

void glutWireDodecahedron(GLdouble radius); void glutSolidDodecahedron(GLdouble radius);

void **glutWireCone**(GLdouble radius, GLdouble height, GLint slices, GLint stacks); void **glutSolidCone**(GLdouble radius, GLdouble height, GLint slices, GLint stacks);

void glutWireTeapot(GLdouble size); void glutSolidTeapot(GLdouble size);

Managing a Background Process

You can specify a function that's to be executed if no other events are pending - for example, when the event loop would otherwise be idle - with **glutIdleFunc()**. This is particularly useful for continuous animation or other background processing.

```
void glutIdleFunc(void (*func)(void));
```

Specifies the function, func, to be executed if no other events are pending. If NULL (zero) is passed in, execution of func is disabled.

Running the Program

After all the setup is completed, GLUT programs enter an event processing loop, glutMainLoop().

void glutMainLoop(void);

Enters the GLUT processing loop, never to return. Registered callback functions will be called when the corresponding events instigate them.



Appendix E Calculating Normal Vectors

This appendix describes how to calculate normal vectors for surfaces. You need to define normals to use the OpenGL lighting facility, which is described in Chapter 5. "Normal Vectors" in Chapter 2 introduces normals and the OpenGL command for specifying them. This appendix goes through the details of calculating them. It has the following major sections:

- "Finding Normals for Analytic Surfaces"
- "Finding Normals from Polygonal Data"

Since normals are perpendicular to a surface, you can find the normal at a particular point on a surface by first finding the flat plane that just touches the surface at that point. The normal is the vector that's perpendicular to that plane. On a perfect sphere, for example, the normal at a point on the surface is in the same direction as the vector from the center of the sphere to that point. For other types of surfaces, there are other, better means for determining the normals, depending on how the surface is specified.

Recall that smooth curved surfaces are approximated by a large number of small flat polygons. If the vectors perpendicular to these polygons are used as the surface normals in such an approximation, the surface appears faceted, since the normal direction is discontinuous across the polygonal boundaries. In many cases, however, an exact mathematical description exists for the surface, and true surface normals can be calculated at every point. Using the true normals improves the rendering considerably, as shown in Figure E-1. Even if you don't have a mathematical description, you can do better than the faceted look shown in the figure. The two major sections in this appendix describe how to calculate normal vectors for these two cases:

- "Finding Normals for Analytic Surfaces" explains what to do when you have a mathematical description of a surface.
- "Finding Normals from Polygonal Data" covers the case when you have only the polygonal data to describe a surface.



Figure E-1 : Rendering with Polygonal Normals vs. True Normals

Finding Normals for Analytic Surfaces

Analytic surfaces are smooth, differentiable surfaces that are described by a mathematical equation (or set of equations). In many cases, the easiest surfaces to find normals for are analytic surfaces for which you have an explicit definition in the following form:

$$\mathbf{V}(s,t) = [\mathbf{X}(s,t) \mathbf{Y}(s,t) \mathbf{Z}(s,t)]$$

where s and t are constrained to be in some domain, and X, Y, and Z are differentiable functions of two variables. To calculate the normal, find

$$\frac{\partial V}{\partial s}$$
 and $\frac{\partial V}{\partial t}$

which are vectors tangent to the surface in the s and t directions. The cross product

$$\frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t}$$

is perpendicular to both and, hence, to the surface. The following shows how to calculate the cross product of two vectors. (Watch out for the degenerate cases where the cross product has zero length!)

$$\begin{bmatrix} \nu_x \nu_y \nu_z \end{bmatrix} \times \begin{bmatrix} w_x w_y w_z \end{bmatrix} = \begin{bmatrix} (\nu_y w_z - w_y \nu_z) (w_x \nu_z - \nu_x w_z) (\nu_x w_y - w_x \nu_y) \end{bmatrix}$$

You should probably normalize the resulting vector. To normalize a vector [x y z], calculate its length

Length = $\sqrt{x^2 + y^2 + z^2}$

and divide each component of the vector by the length.

As an example of these calculations, consider the analytic surface

 $V(s,t) = [s_2 t_3 3 - s_t]$

From this we have

$$\frac{\partial V}{\partial s} = \begin{bmatrix} 2s \ 0 \ -t \end{bmatrix}, \frac{\partial V}{\partial t} = \begin{bmatrix} 0 \ 3t^2 \ -s \end{bmatrix}, \text{ and } \frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t} = \begin{bmatrix} -3t^3 \ 2s^2 \ 6st^2 \end{bmatrix}$$

So, for example, when s=1 and t=2, the corresponding point on the surface is (1, 8, 1), and the vector (-24, 2, 24) is perpendicular to the surface at that point. The length of this vector is 34, so the unit normal vector is (-24/34, 2/34, 24/34) = (-0.70588, 0.058823, 0.70588).

For analytic surfaces that are described implicitly, as $\mathbf{F}(x, y, z) = 0$, the problem is harder. In some cases, you can solve for one of the variables, say $z = \mathbf{G}(x, y)$, and put it in the explicit form given previously:

 $\mathbf{V}(s, t) = [s \ t \ \mathbf{G}(s, t)]$

Then continue as described earlier.

If you can't get the surface equation in an explicit form, you might be able to make use of the fact that the normal vector is given by the gradient

 $VF = \begin{bmatrix} F & F & F \\ \hline x & y & z \end{bmatrix}$

evaluated at a particular point (x, y, z). Calculating the gradient might be easy, but finding a point that lies on the surface can be difficult. As an example of an implicitly defined analytic function, consider the equation of a sphere of radius 1 centered at the origin:

 $x^2 + y^2 + z^2 - 1 = 0$)

This means that

$$\mathbf{F}(x, y, z) = x2 + y2 + z2 - 1$$

which can be solved for z to yield

$$z = \pm \sqrt{1 - x^2 - y^2}$$

Thus, normals can be calculated from the explicit form

 $\mathbf{V}(s, t) = \begin{bmatrix} s \ t \ \sqrt{1 - s^2 - t^2} \end{bmatrix}$

as described previously.

If you could not solve for z, you could have used the gradient

 $VF = \begin{bmatrix} 2x & 2y & 2z \end{bmatrix}$

as long as you could find a point on the surface. In this case, it's not so hard to find a point - for example, (2/3, 1/3, 2/3) lies on the surface. Using the gradient, the normal at this point is (4/3, 2/3, 4/3). The unit-length normal is (2/3, 1/3, 2/3), which is the same as the point on the surface, as expected.

Finding Normals from Polygonal Data

As mentioned previously, you often want to find normals for surfaces that are described with polygonal data such that the surfaces appear smooth rather than faceted. In most cases, the easiest way for you to do this (though it might not be the most efficient way) is to calculate the normal vectors for each of the polygonal facets and then to average the normals for neighboring facets. Use the averaged normal for the vertex that the neighboring facets have in common. Figure E-2 shows a surface and its polygonal approximation. (Of course, if the polygons represent the exact surface and aren't merely an approximation - if you're drawing a cube or a cut diamond, for example - don't do the averaging. Calculate the normal for each facet as described in the following paragraphs, and use that same normal for each vertex of the facet.)

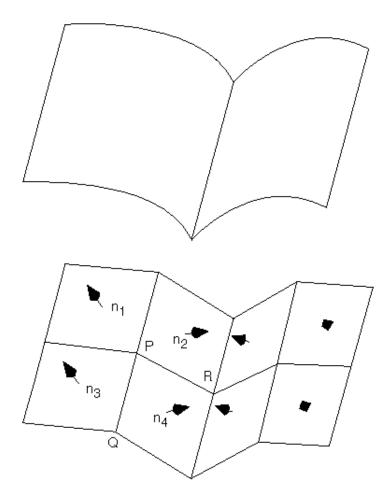


Figure E-2 : Averaging Normal Vectors

To find the normal for a flat polygon, take any three vertices v1, v2, and v3 of the polygon that do not lie in a straight line. The cross product

 $[v1 - v2] \times [v2 - v3]$

is perpendicular to the polygon. (Typically, you want to normalize the resulting vector.) Then you need to average the normals for adjoining facets to avoid giving too much weight to one of them. For instance, in the example shown in Figure E-2, if n1, n2, n3, and n4 are the normals for the four polygons meeting at point P, calculate n1+n2+n3+n4 and then normalize it. (You can get a better average if you weight the normals by the size of the angles at the shared intersection.) The resulting

vector can be used as the normal for point P.

+ +

Sometimes, you need to vary this method for particular situations. For instance, at the boundary of a surface (for example, point Q in Figure E-2), you might be able to choose a better normal based on your knowledge of what the surface should look like. Sometimes the best you can do is to average the polygon normals on the boundary as well. Similarly, some models have some smooth parts and some sharp corners (point R is on such an edge in Figure E-2). In this case, the normals on either side of the crease shouldn't be averaged. Instead, polygons on one side of the crease should be drawn with one normal, and polygons on the other side with another.

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Appendix F Homogeneous Coordinates and Transformation Matrices

This appendix presents a brief discussion of homogeneous coordinates. It also lists the form of the transformation matrices used for rotation, scaling, translation, perspective projection, and orthographic projection. These topics are introduced and discussed in Chapter 3. For a more detailed discussion of these subjects, see almost any book on three-dimensional computer graphics - for example, *Computer Graphics: Principles and Practice* by Foley, van Dam, Feiner, and Hughes (Reading, MA: Addison-Wesley, 1990) - or a text on projective geometry - for example, *The Real Projective Plane*, by H. S. M. Coxeter, 2nd ed. (Cambridge: Cambridge University Press, 1961). In the discussion that follows, the term homogeneous coordinates always means three-dimensional homogeneous coordinates, although projective geometries exist for all dimensions.

This appendix has the following major sections:

- "Homogeneous Coordinates"
- "Transformation Matrices"

Homogeneous Coordinates

OpenGL commands usually deal with two- and three-dimensional vertices, but in fact all are treated internally as three-dimensional homogeneous vertices comprising four coordinates. Every column vector (*x*, *y*, *z*, *w*)T represents a homogeneous vertex if at least one of its elements is nonzero. If the real number *a* is nonzero, then (*x*, *y*, *z*, *w*)T and (*ax*, *ay*, *az*, *aw*)T represent the same homogeneous vertex. (This is just like fractions: x/y = (ax)/(ay).) A three-dimensional euclidean space point (*x*, *y*, *z*)T becomes the homogeneous vertex with coordinates (*x*, *y*, *z*, 1.0)T, and the two-dimensional euclidean point (*x*, *y*)T becomes (*x*, *y*, 0.0, 1.0)T.

As long as *w* is nonzero, the homogeneous vertex (*x*, *y*, *z*, *w*)T corresponds to the three-dimensional point (x/w, y/w, z/w)T. If w = 0.0, it corresponds to no euclidean point, but rather to some idealized "point at infinity." To understand this point at infinity, consider the point (1, 2, 0, 0), and note that the sequence of points (1, 2, 0, 1), (1, 2, 0, 0.01), and (1, 2.0, 0.0, 0.0001), corresponds to the euclidean points (1, 2), (100, 200), and (10000, 20000). This sequence represents points rapidly moving toward infinity along the line 2x = y. Thus, you can think of (1, 2, 0, 0) as the point at infinity in the direction of that line.

Note: OpenGL might not handle homogeneous clip coordinates with w < 0 correctly. To be sure that your code is portable to all OpenGL systems, use only nonnegative w values.

Transforming Vertices

Vertex transformations (such as rotations, translations, scaling, and shearing) and projections (such as perspective and orthographic) can all be represented by applying an appropriate 4×4 matrix to the coordinates representing the vertex. If **v** represents a homogeneous vertex and **M** is a 4×4 transformation matrix, then **Mv** is the image of **v** under the transformation by **M**. (In computer-graphics applications, the transformations used are usually nonsingular - in other words, the matrix **M** can be inverted. This isn't required, but some problems arise with nonsingular transformations.)

After transformation, all transformed vertices are clipped so that *x*, *y*, and *z* are in the range [-&ohgr; w] (assuming w > 0). Note that this range corresponds in euclidean space to [-1.0, 1.0].

Transforming Normals

Normal vectors aren't transformed in the same way as vertices or position vectors. Mathematically, it's better to think of normal vectors not as vectors, but as planes perpendicular to those vectors. Then, the transformation rules for normal vectors are described by the transformation rules for perpendicular planes.

A homogeneous plane is denoted by the row vector (a, b, c, d), where at least one of a, b, c, or d is nonzero. If q is a nonzero real number, then (a, b, c, d) and (qa, qb, qc, qd) represent the same plane. A point (x, y, z, w)T is on the plane (a, b, c, d) if ax+by+cz+dw = 0. (If w = 1, this is the standard description of a euclidean plane.) In order for (a, b, c, d) to represent a euclidean plane, at least one of a, b, or c must be nonzero. If they're all zero, then (0, 0, 0, d) represents the "plane at infinity," which contains all the "points at infinity."

If **p** is a homogeneous plane and **v** is a homogeneous vertex, then the statement "**v** lies on plane **p**" is written mathematically as $\mathbf{pv} = 0$, where \mathbf{pv} is normal matrix multiplication. If **M** is a nonsingular vertex transformation (that is, a 4×4 matrix that has an inverse **M**-1), then $\mathbf{pv} = 0$ is equivalent to \mathbf{pM} -1 $\mathbf{Mv} = 0$, so \mathbf{Mv} lies on the plane \mathbf{pM} -1. Thus, \mathbf{pM} -1 is the image of the plane under the vertex transformation **M**.

If you like to think of normal vectors as vectors instead of as the planes perpendicular to them, let \mathbf{v} and \mathbf{n} be vectors such that \mathbf{v} is perpendicular to \mathbf{n} . Then, $\mathbf{nTv} = 0$. Thus, for an arbitrary nonsingular transformation \mathbf{M} , \mathbf{nTM} -1 $\mathbf{Mv} = 0$, which means that \mathbf{nTM} -1 is the transpose of the transformed normal vector. Thus, the transformed normal vector is (\mathbf{M} -1) \mathbf{Tn} . In other words, normal vectors are transformed by the inverse transpose of the transformation that transforms points. Whew!

Transformation Matrices

Although any nonsingular matrix **M** represents a valid projective transformation, a few special matrices are particularly useful. These matrices are listed in the following subsections.

Translation

The call **glTranslate***(x, y, z) generates **T**, where

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

The call **glScale***(*x*, *y*, *z*) generates S, where

$$S = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } S^{-1} = \begin{bmatrix} \frac{1}{x} & 0 & 0 & 0 \\ 0 & \frac{1}{y} & 0 & 0 \\ 0 & 0 & \frac{1}{z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that S-1 is defined only if *x*, *y*, and *z* are all nonzero.

Rotation

The call **glRotate***(*a*, *x*, *y*, *z*) generates R as follows:

Let v = (x, y, z)T, and u = v/||v|| = (x', y', z')T.

Also let

$$S = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix} \text{ and } M = uu^{T} + (\cos \alpha) (I - uu^{T}) + (\sin \alpha) S$$

Then

$$R = \begin{bmatrix} m \ m \ m \ 0 \\ m \ m \ m \ 0 \\ m \ m \ m \ 0 \\ 0 \ 0 \ 1 \end{bmatrix}$$
 where *m* represents elements from M, which is a 3x3 matrix

The **R** matrix is always defined. If x=y=z=0, then **R** is the identity matrix. You can obtain the inverse of **R**, **R-1**, by substituting - &*agr*; for *a*, or by transposition.

The **glRotate***() command generates a matrix for rotation about an arbitrary axis. Often, you're rotating about one of the coordinate axes; the corresponding matrices are as follows:

$$glRotate*(a, 1, 0, 0): \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$glRotate*(a, 0, 1, 0): \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$glRotate*(a, 0, 0, 1): \begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ -\sin a & \cos a & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As before, the inverses are obtained by transposition.

Perspective Projection

The call **glFrustum**(l, r, b, t, n, f) generates **R**, where

$$R = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0\\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0\\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n}\\ 0 & 0 & -1 & 0 \end{bmatrix} \text{ and } R^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n}\\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n}\\ 0 & 0 & 0 & -1\\ 0 & 0 & 0 & -1\\ 0 & 0 & \frac{-(f-n)}{2fn} & \frac{f+n}{2fn} \end{bmatrix}$$

R is defined as long as l ≠ r, t ≠ b, and n ≠ f.

Orthographic Projection

The call **glOrtho**(l, r, b, t, n, f) generates **R**, where

$$R = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } R^{-1} = \begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R is defined as long as l ≠ r, t ≠ b, and n ≠ f.



(Addison-Wesley Publishing Company)

Appendix G Programming Tips

This appendix lists some tips and guidelines that you might find useful. Keep in mind that these tips are based on the intentions of the designers of the OpenGL, not on any experience with actual applications and implementations! This appendix has the following major sections:

- "OpenGL Correctness Tips"
- "OpenGL Performance Tips"
- "GLX Tips"

OpenGL Correctness Tips

- Perform error checking often. Call **glGetError**() at least once each time the scene is rendered to make certain error conditions are noticed.
- Do not count on the error behavior of an OpenGL implementation it might change in a future release of OpenGL. For example, OpenGL 1.1 ignores matrix operations invoked between **glBegin()** and **glEnd()** commands, but a future version might not. Put another way, OpenGL error semantics may change between upward-compatible revisions.
- If you need to collapse all geometry to a single plane, use the projection matrix. If the modelview matrix is used, OpenGL features that operate in eye coordinates (such as lighting and application-defined clipping planes) might fail.
- Do not make extensive changes to a single matrix. For example, do not animate a rotation by continually calling **glRotate***() with an incremental angle. Rather, use **glLoadIdentity**() to initialize the given matrix for each frame, then call **glRotate***() with the desired complete angle for that frame.
- Count on multiple passes through a rendering database to generate the same pixel fragments only if this behavior is guaranteed by the invariance rules established for a compliant OpenGL implementation. (See Appendix H for details on the invariance rules.) Otherwise, a different set of fragments might be generated.
- Do not expect errors to be reported while a display list is being defined. The commands within a display list generate errors only when the list is executed.
- Place the near frustum plane as far from the viewpoint as possible to optimize the operation

of the depth buffer.

- Call glFlush() to force all previous OpenGL commands to be executed. Do not count on glGet*() or glIs*() to flush the rendering stream. Query commands flush as much of the stream as is required to return valid data but don't guarantee completing all pending rendering commands.
- Turn dithering off when rendering predithered images (for example, when **glCopyPixels**() is called).
- Make use of the full range of the accumulation buffer. For example, if accumulating four images, scale each by one-quarter as it's accumulated.
- If exact two-dimensional rasterization is desired, you must carefully specify both the orthographic projection and the vertices of primitives that are to be rasterized. The orthographic projection should be specified with integer coordinates, as shown in the following example:

```
gluOrtho2D(0, width, 0, height);
```

where *width* and *height* are the dimensions of the viewport. Given this projection matrix, polygon vertices and pixel image positions should be placed at integer coordinates to rasterize predictably. For example, **glRecti**(0, 0, 1, 1) reliably fills the lower left pixel of the viewport, and **glRasterPos2i**(0, 0) reliably positions an unzoomed image at the lower left of the viewport. Point vertices, line vertices, and bitmap positions should be placed at half-integer locations, however. For example, a line drawn from (x1, 0.5) to (x2, 0.5) will be reliably rendered along the bottom row of pixels into the viewport, and a point drawn at (0.5, 0.5) will reliably fill the same pixel as **glRecti**(0, 0, 1, 1).

An optimum compromise that allows all primitives to be specified at integer positions, while still ensuring predictable rasterization, is to translate x and y by 0.375, as shown in the following code fragment. Such a translation keeps polygon and pixel image edges safely away from the centers of pixels, while moving line vertices close enough to the pixel centers.

```
glViewport(0, 0, width, height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, width, 0, height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.375, 0.375, 0.0);
/* render all primitives at integer positions */
```

- Avoid using negative *w* vertex coordinates and negative *q* texture coordinates. OpenGL might not clip such coordinates correctly and might make interpolation errors when shading primitives defined by such coordinates.
- Do not assume the precision of operations, based upon the data type of parameters to OpenGL commands. For example, if you are using **glRotated**(), you should not assume that geometric processing pipeline operates with double-precision floating point. It is possible that the parameters to **glRotated**() are converted to a different data type before processing.

OpenGL Performance Tips

- Use **glColorMaterial**() when only a single material property is being varied rapidly (at each vertex, for example). Use **glMaterial**() for infrequent changes, or when more than a single material property is being varied rapidly.
- Use **glLoadIdentity**() to initialize a matrix, rather than loading your own copy of the identity matrix.
- Use specific matrix calls such as **glRotate***(), **glTranslate***(), and **glScale***() rather than composing your own rotation, translation, or scale matrices and calling **glMultMatrix**().
- Use query functions when your application requires just a few state values for its own computations. If your application requires several state values from the same attribute group, use **glPushAttrib**() and **glPopAttrib**() to save and restore them.
- Use display lists to encapsulate potentially expensive state changes.
- Use display lists to encapsulate the rendering calls of rigid objects that will be drawn repeatedly.
- Use texture objects to encapsulate texture data. Place all the **glTexImage***() calls (including mipmaps) required to completely specify a texture and the associated **glTexParameter***() calls (which set texture properties) into a texture object. Bind this texture object to select the texture.
- If the situation allows it, use **gl*TexSubImage**() to replace all or part of an existing texture image rather than the more costly operations of deleting and creating an entire new image.
- If your OpenGL implementation supports a high-performance working set of resident textures, try to make all your textures resident; that is, make them fit into the high-performance texture memory. If necessary, reduce the size or internal format resolution of your textures until they all fit into memory. If such a reduction creates intolerably fuzzy textured objects, you may give some textures lower priority, which will, when push comes to shove, leave them out of the working set.
- Use evaluators even for simple surface tessellations to minimize network bandwidth in client-server environments.
- Provide unit-length normals if it's possible to do so, and avoid the overhead of GL_NORMALIZE. Avoid using **glScale***() when doing lighting because it almost always requires that GL_NORMALIZE be enabled.
- Set **glShadeModel**() to GL_FLAT if smooth shading isn't required.
- Use a single **glClear**() call per frame if possible. Do not use **glClear**() to clear small subregions of the buffers; use it only for complete or near-complete clears.
- Use a single call to **glBegin**(GL_TRIANGLES) to draw multiple independent triangles rather than calling **glBegin**(GL_TRIANGLES) multiple times, or calling **glBegin**(GL_POLYGON).

Even if only a single triangle is to be drawn, use GL_TRIANGLES rather than GL_POLYGON. Use a single call to **glBegin**(GL_QUADS) in the same manner rather than calling **glBegin**(GL_POLYGON) repeatedly. Likewise, use a single call to **glBegin**(GL_LINES) to draw multiple independent line segments rather than calling **glBegin**(GL_LINES) multiple times.

- Some OpenGL implementations benefit from storing vertex data in vertex arrays. Use of vertex arrays reduces function call overhead. Some implementations can improve performance by batch processing or reusing processed vertices.
- In general, use the vector forms of commands to pass precomputed data, and use the scalar forms of commands to pass values that are computed near call time.
- Avoid making redundant mode changes, such as setting the color to the same value between each vertex of a flat-shaded polygon.
- Be sure to disable expensive rasterization and per-fragment operations when drawing or copying images. OpenGL will even apply textures to pixel images if asked to!
- Unless absolutely needed, avoid having different front and back polygon modes.

GLX Tips

- Use **glXWaitGL**() rather than **glFinish**() to force X rendering commands to follow GL rendering commands.
- Likewise, use **glXWaitX()** rather than **XSync()** to force GL rendering commands to follow X rendering commands.
- Be careful when using **glXChooseVisual()**, because boolean selections are matched exactly. Since some implementations won't export visuals with all combinations of boolean capabilities, you should call **glXChooseVisual()** several times with different boolean values before you give up. For example, if no single-buffered visual with the required characteristics is available, check for a double-buffered visual with the same capabilities. It might be available, and it's easy to use.

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)

Appendix H OpenGL Invariance

OpenGL is not a pixel-exact specification. It therefore doesn't guarantee an exact match between images produced by different OpenGL implementations. However, OpenGL does specify exact matches, in some cases, for images produced by the same implementation. This appendix describes the invariance rules that define these cases.

The obvious and most fundamental case is repeatability. A conforming OpenGL implementation generates the same results each time a specific sequence of commands is issued from the same initial conditions. Although such repeatability is useful for testing and verification, it's often not useful to application programmers, because it's difficult to arrange for equivalent initial conditions. For example, rendering a scene twice, the second time after swapping the front and back buffers, doesn't meet this requirement. So repeatability can't be used to guarantee a stable, double-buffered image.

A simple and useful algorithm that counts on invariant execution is erasing a line by redrawing it in the background color. This algorithm works only if rasterizing the line results in the same fragment *x*, *y* pairs being generated in both the foreground and background color cases. OpenGL requires that the coordinates of the fragments generated by rasterization be invariant with respect to framebuffer contents, which color buffers are enabled for drawing, the values of matrices other than those on the top of the matrix stacks, the scissor parameters, all writemasks, all clear values, the current color, index, normal, texture coordinates, and edge-flag values, the current raster color, raster index, and raster texture coordinates, and the material properties. It is further required that exactly the same fragments be generated, including the fragment color values, when framebuffer contents, color buffer enables, matrices other than those on the top of the matrix stacks, the scissor parameters, writemasks, or clear values differ.

OpenGL further suggests, but doesn't require, that fragment generation be invariant with respect to the matrix mode, the depths of the matrix stacks, the alpha test parameters (other than alpha test enable), the stencil parameters (other than stencil enable), the depth test parameters (other than depth test enable), the blending parameters (other than enable), the logical operation (but not logical operation enable), and the pixel-storage and pixel-transfer parameters. Because invariance with respect to several enables isn't recommended, you should use other parameters to disable functions when invariant rendering is required. For example, to render invariantly with blending enabled and disabled, set the blending parameters to GL_ONE and GL_ZERO to disable blending rather than calling **glDisable**(GL_BLEND). Alpha testing, stencil testing, depth testing, and the logical operation all can be disabled in this manner.

Finally, OpenGL requires that per-fragment arithmetic, such as blending and the depth test, is invariant to all OpenGL state except the state that directly defines it. For example, the only OpenGL parameters that affect how the arithmetic of blending is performed are the source and destination blend parameters and the blend enable parameter. Blending is invariant to all other state changes. This invariance holds for the scissor test, the alpha test, the stencil test, the depth test, blending, dithering, logical operations, and buffer writemasking.

As a result of all these invariance requirements, OpenGL can guarantee that images rendered into different color buffers, either simultaneously or separately using the same command sequence, are pixel identical. This holds for all the color buffers in the framebuffer or all the color buffers in an off-screen buffer, but it isn't guaranteed between the framebuffer and off-screen buffers.

+ +

OpenGL Programming Guide (Addison-Wesley Publishing Company)