



present

JEFF MOLOFEE'S

OpenGL

WINDOWS TUTORIAL



## *Tutorial Index*



### **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.

### **Setting Up OpenGL In Windows:**



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.

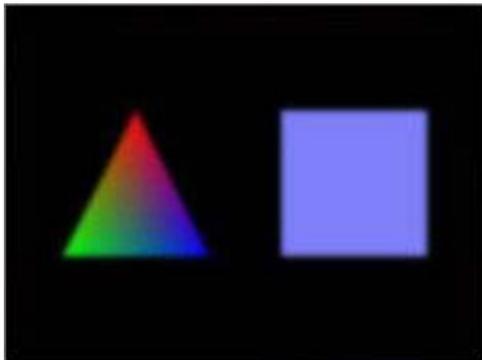
### **Your First Polygon:**

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.

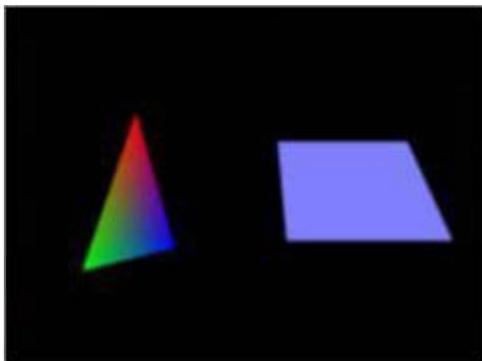
Once you've read through this chapter, you should understand the X axis, Y axis and Z axis. You will learn about translation left, right, up, down, into and out of the screen. You should understand how to place an object on the screen exactly where you want it to be. You will also learn a bit about the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

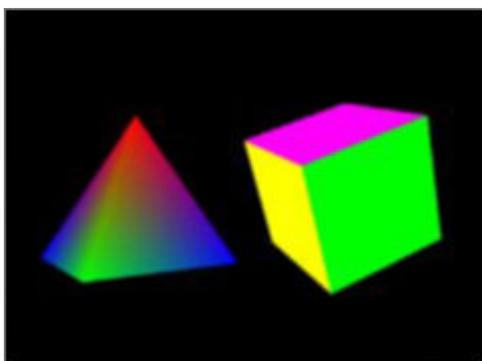
Color adds a lot to an OpenGL project. By understanding both flat and smooth coloring, you can greatly enhance the way your OpenGL demos look.



### 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 objects 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 to map a bitmap image onto the



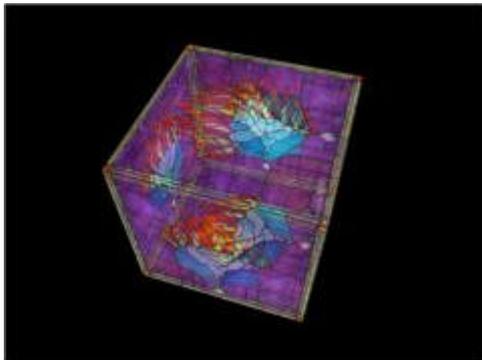
six side of a cube. We'll use the GL code from lesson one to create this project. It's easier to start with an empty GL window than to modify the last tutorial.

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.

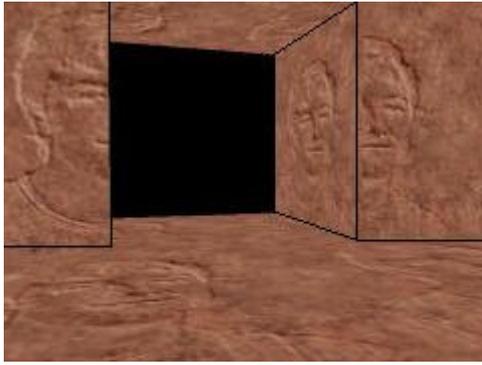
### **Moving Bitmaps In 3D Space:**



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](#).

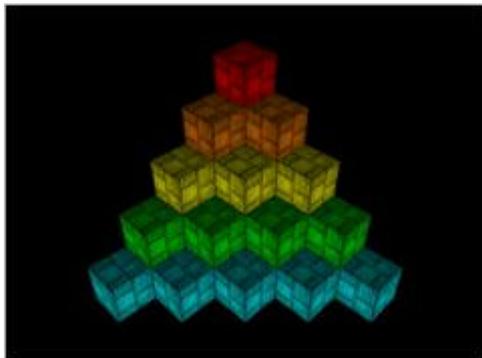


In this lesson you will learn how to load a 3D world from a data file, and move through the 3D world. The code is made using lesson 1 code, however, the tutorial web page only explains the NEW code used to load the 3D scene, and move around inside the 3D world. Download the VC++ code, and follow through 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 brought to you by Bosco. The same guy that created the totally cool mini demo called worthless. He enjoyed everyone's 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 a lot 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 your 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. Bitmap fonts 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.



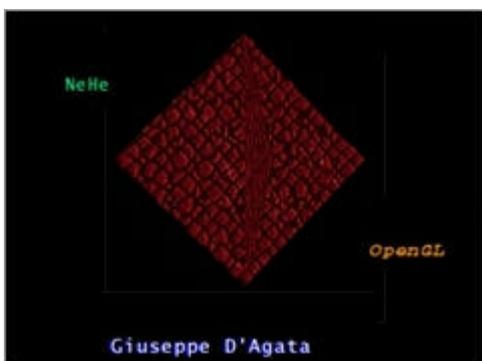
### **Texture Mapped Fonts:**

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 metallic looking object that reflects anything from a pattern to a picture.



### **\* Cool Looking Fog:**

This tutorial 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!

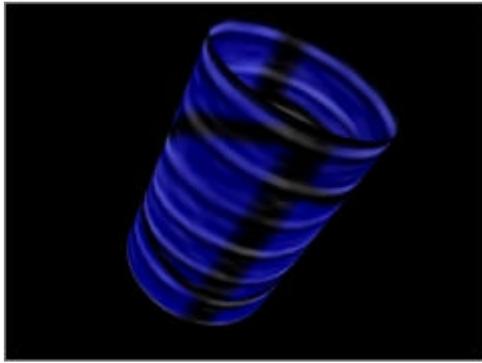


### **\* 2D Texture Font:**

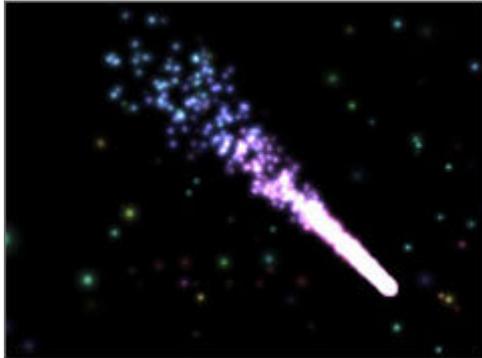
The original version of this tutorial was written by Giuseppe D'Agata. In this tutorial you will learn how to write any character or phrase you want to the screen using texture mapped quads. You will learn how to read one of 256 different characters from a 256x256 texture map, and finally I will show 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 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 possibilities!

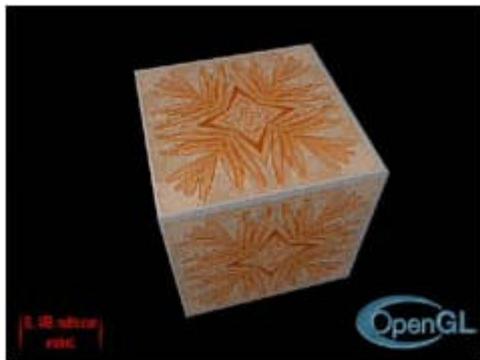


### Lines, Antialiasing, Timing, Ortho View And Simple Sounds:

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 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 bump-mapping. 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 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 a lot of time making the tutorial unique (reading textures from the data file, etc), and I spent a lot of time tweaking things, and making the HTML look pretty. If you enjoy this tutorial let him know!

#### **\* Sphere Mapping Quadratics In OpenGL:**



This tutorial 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 a lot 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!

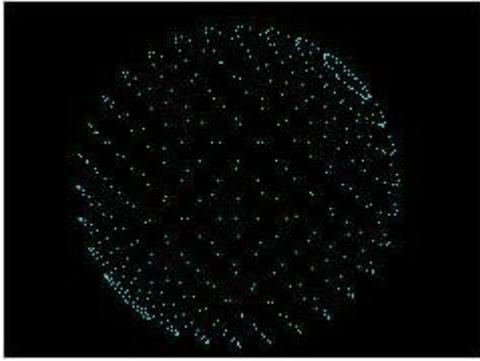
#### **Tokens, Extensions, Scissor Testing And TGA Loading:**



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 glAUX library. Something I'm sure a lot of you guys will appreciate!

#### **\* Morphing & Loading Objects From A File:**

This tutorial 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!

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> // Header
#include <gl\glu.h>
#include <gl\glaux.h>
```

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; // Privat
HGLRC          hRC=NULL; // Perman
HWND           hWnd=NULL;
HINSTANCE hInstance; // 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; // Fullsc
```

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); // Declar
```

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);                             // Select
    glLoadIdentity();                                       // 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 won't worry about the value for now.

```
int InitGL(GLvoid)                                         // All Se
{
```

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.

```
glShadeModel(GL_SMOOTH); // Enable
```

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 won't 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); // Enable
glDepthFunc(GL_LEQUAL);
```

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 `glLoadIdentity()` 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 won't 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);           // Clear '
    glLoadIdentity();                                             // Reset '
    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)                                     // Proper
{
```

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
    }
```

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 I
{
```

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.

```

if (hWnd && !DestroyWindow(hWnd)) // Are We
{
    MessageBox(NULL,"Could Not Release hWnd.", "SHUTDOWN ERROR",MB_OK | MB_ICO
    hWnd=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".

```

if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To U
{
    MessageBox(NULL,"Could Not Unregister Class.", "SHUTDOWN ERROR",MB_OK | MB
    hInstance=NULL;
}
}

```

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; // Window
```

**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 dwExStyle;
DWORD dwStyle; // Window
```

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; // Grabs l
WindowRect.left=(long)0; // Set Le
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; // Set Th
```

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.

```

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 Bac
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))                       // Attempt
{
    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
memset(&dmScreenSettings, 0, sizeof(dmScreenSettings)); // Makes
dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size O
dmScreenSettings.dmPelsWidth  = width;         // Select
dmScreenSettings.dmPelsHeight = height;       // Select
dmScreenSettings.dmBitsPerPel = bits;
dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

```

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.

```
fullscreen=FALSE; // Select
}
else
{
```

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; // Window
        dwStyle=WS_POPUP; // Window
        ShowCursor(FALSE); // Hide M
    }
    else
    {

```

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, // Extend
                              "OpenGL", // Class I
                              title,
                              WS_CLIPSIBLINGS | // Require
                              WS_CLIPCHILDREN | // Require
                              dwStyle, // Select
                              0, 0,
                              WindowRect.right-WindowRect.left, // Calcul.
                              WindowRect.bottom-WindowRect.top, // Calcul.
                              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, // Color :
    0, 0, 0, 0, 0, 0,
    0,
    0,
    0,
    0, 0, 0, 0,
    16,
    0,
    0,
    PFD_MAIN_PLANE,
    0,
    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 (!(PixelFormat=ChoosePixelFormat(hDC,&pfd))
{
    KillGLWindow();
    MessageBox(NULL,"Can't Find A Suitable PixelFormat.,"ERROR",MB_OK|MB_ICO
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 (!(hRC=wglCreateContext(hDC)) // Are We
{
    KillGLWindow();
    MessageBox(NULL,"Can't Create A GL Rendering Context.,"ERROR",MB_OK|MB_I
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(!wglMakeCurrent(hDC,hRC)
{
    KillGLWindow();
    MessageBox(NULL,"Can't Activate The GL Rendering Context.,"ERROR",MB_OK|
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); // Show T
SetForegroundWindow(hWnd); // Slight
SetFocus(hWnd);
ReSizeGLScene(width, height);
```

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` (everything's OK) or `FALSE` (something's 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 registered 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:
{
    if (!HIWORD(wParam))
    {
        active=TRUE;
    }
    else
    {
        active=FALSE;
    }

    return 0;
}
// Watch :
// 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;
    }
    break;
}
// Prevent

```

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.

```

case WM_CLOSE:
{
    PostQuitMessage(0);
    return 0;
}
// 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.

```

case WM_KEYDOWN:
// Is A Ke

```

```

    {
        keys[wParam] = TRUE;
        return 0; // Jump B.
    }

```

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; // Jump B.
    }

```

Whenever we resize our window **uMsg** will eventually become the message WM\_SIZE. We read the LOWORD and HIWORD values of **lParam** 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.

```

    case WM_SIZE:
    {
        ReSizeGLScene(LOWORD(lParam),HIWORD(lParam)); // LoWord
        return 0; // Jump B.
    }
}

```

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
    HINSTANCEhPrevInstance, // 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 (msg.message==WM_QUIT)                // Have W
        {
            done=TRUE;
        }
        else
        {

```

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.

```

                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
        else
        {

```

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.

```

        // Draw The Scene. Watch For ESC Key And Quit Messages From Dra
        if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Updati
        {
            done=TRUE;
        }
        else
        {

```

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
            }

```

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.

```

        if (keys[VK_F1]) // Is F1 :
        {
            keys[VK_F1]=FALSE; // If So I
            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 Herboth](#) )
- \* [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 kWindowWidth      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](mailto: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 necessary 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 little 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 necessary for your 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 necessary patches for your OS version and install them. BTW. You need root access to do this. Ask your 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/sparc64/lib
> 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/sparc64/lib
```

Verify that everything is correct:

```
> echo $LD_LIBRARY_PATH
/lib:/usr/lib:/usr/openwin/lib:/usr/dt/lib:/usr/local/lib:/usr/local/sparc64/lib:/usr/local/sparc64/lib:/usr/local/sparc64/lib:/usr/local/sparc64/lib:/usr/local/sparc64/lib:/usr/local/sparc64/lib
```

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!](#)

## 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
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen
    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);                             // Move L
```

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()`.

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); // Drawin
    glVertex3f( 0.0f, 1.0f, 0.0f); // Top
    glVertex3f(-1.0f,-1.0f, 0.0f); // Bottom
    glVertex3f( 1.0f,-1.0f, 0.0f); // Bottom
glEnd(); // Finished Drawing
```

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); // Move R
```

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 Ri
    glVertex3f( 1.0f,-1.0f, 0.0f); // Bottom
    glVertex3f(-1.0f,-1.0f, 0.0f); // Bottom
glEnd(); // Done Drawing Th
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 F1 Being Pres
{
    keys[VK_F1]=FALSE; // If So Make Key 1
    KillGLWindow(); // Kill O
    fullscreen=!fullscreen; // Toggle
    // Recreate Our OpenGL Window ( Modified )
    if (!CreateGLWindow("NeHe's First Polygon Tutorial",640
    {
        return 0; // Quit If Window 1
    }
}
```

```
}
```

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 Herboth](#) )
- \* 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 Modelview

    glTranslatef(-1.5f,0.0f,-6.0f);                         // Left 1.5 Then In

    glBegin(GL_TRIANGLES);                                   // Begin Drawing Tri
```

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 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);                             // Set The Color To Green
    glVertex3f(-1.0f,-1.0f, 0.0f);                          // Left And Down On
```

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);           // Set The Color To Blue
        glVertex3f( 1.0f,-1.0f, 0.0f);      // Right And Down (
glEnd();                                     // Done Drawing A Triangle

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

```

        glColor3f(0.5f,0.5f,1.0f);         // Set The Color To Blue One
glBegin(GL_QUADS);                         // Start Drawing Quads
        glVertex3f(-1.0f, 1.0f, 0.0f);    // Left And Up 1 U
        glVertex3f( 1.0f, 1.0f, 0.0f);    // Right And Up 1 U
        glVertex3f( 1.0f,-1.0f, 0.0f);    // Left And Up One
        glVertex3f(-1.0f,-1.0f, 0.0f);    // Left And Up One
glEnd();                                    // Done Drawing A Quad
return TRUE;                                // 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 Fullscreen
            // Recreate Our OpenGL Window ( Modified )
            if (!CreateGLWindow("NeHe's Color Tutorial",640,480,16,
            {
                return 0;                   // Quit If Window Was Not Cr
            }
        }
}

```

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 Herboth](#) )
- \* 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 <windows.h> // Header File For
#include <gl\gl.h> // Header File For The OpenC
#include <gl\glu.h> // Header File For
#include <gl\glaux.h> // Header File For

HDC hDC=NULL; // Private GDI Device Contex
HGLRC hRC=NULL; // Permanent Rendering Contex
HWND hWnd=NULL; // Holds Our Window

bool keys[256]; // Array Used For
bool active=TRUE; // Window Active Fl
bool fullscreen=TRUE; // Fullscreen Flag Set To TR

GLfloat rtri; // Angle For The Tri
GLfloat rquad; // Angle For The Qu
```

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 S
```

The next line of code is new. `glRotatef(Angle,Xvector,Yvector,Zvector)` is responsible for rotating the object around an axis. You will get a lot 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 Triangle
```

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 its Y axis causing it to spin left to right.

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

You'll notice in the code below, that we've added another `glLoadIdentity()`. 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 `glLoadIdentity()` 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 don't 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(); // Reset The Current Modelview
glTranslatef(1.5f,0.0f,-6.0f); // Move Right 1.5 Units
glRotatef(rquad,1.0f,0.0f,0.0f); // Rotate The Quad On The X
```

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.

```
glColor3f(0.5f,0.5f,1.0f); // Set The Color To A Nice Blue
glBegin(GL_QUADS); // Start Drawing A Quad
    glVertex3f(-1.0f, 1.0f, 0.0f); // Top Left Of The Quad
    glVertex3f( 1.0f, 1.0f, 0.0f); // Top Right Of The Quad
    glVertex3f( 1.0f,-1.0f, 0.0f); // Bottom Right Of The Quad
    glVertex3f(-1.0f,-1.0f, 0.0f); // Bottom Left Of The Quad
glEnd(); // Done Drawing The Quad
```

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 changing 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; // Increase The Rotation Amount
rquad-=0.15f; // Decrease The Rotation Amount
return TRUE; // 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 Fullscreen
    // Recreate Our OpenGL Window ( Modified )
    if (!CreateGLWindow("NeHe's Rotation Tutorial",640,480,
    {
        return 0; // Quit If Window Was Not Cr
    }
}
```

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 Herboth](#) )
- \* 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();                                     // Reset The View
    glTranslatef(-1.5f,0.0f,-6.0f);                       // Move Left And In
    glRotatef(rtri,0.0f,1.0f,0.0f);                       // Rotate The Pyram
    glBegin(GL_TRIANGLES);                                 // Start Drawing Tri
```

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

```

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);           // 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

```

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.

```

glColor3f(1.0f,0.0f,0.0f);           // Red
glVertex3f( 0.0f, 1.0f, 0.0f);      // Top Of Triangle
glColor3f(0.0f,0.0f,1.0f);         // Blue
glVertex3f(-1.0f,-1.0f,-1.0f);     // Left Of Triangle
glColor3f(0.0f,1.0f,0.0f);         // Green
glVertex3f(-1.0f,-1.0f, 1.0f);     // Right Of Triangle
glEnd();                             // Done Drawing 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 :)

```
glLoadIdentity();
glTranslatef(1.5f,0.0f,-7.0f); // Move Right And Into Screen

glRotatef(rquad,1.0f,1.0f,1.0f); // Rotate The Cube On X, Y & Z Axis

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 one 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 The Color To Blue
glVertex3f( 1.0f, 1.0f,-1.0f); // Top Right Of The Quad (Top)
glVertex3f(-1.0f, 1.0f,-1.0f); // Top Left Of The Quad (Top)
glVertex3f(-1.0f, 1.0f, 1.0f); // Bottom Left Of The Quad (Top)
glVertex3f( 1.0f, 1.0f, 1.0f); // Bottom Right Of The Quad (Top)
```

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); // Set The Color To Orange
glVertex3f( 1.0f,-1.0f, 1.0f); // Top Right Of The Quad (Bottom)
glVertex3f(-1.0f,-1.0f, 1.0f); // Top Left Of The Quad (Bottom)
glVertex3f(-1.0f,-1.0f,-1.0f); // Bottom Left Of The Quad (Bottom)
glVertex3f( 1.0f,-1.0f,-1.0f); // Bottom Right Of The Quad (Bottom)
```

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
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);         // Set The Color To Blue
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

```

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);         // Set The Color To Violet
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
glEnd();                             // Done Drawing The Quad

rtri+=0.2f;                           // Increase The Rot
rquad-=0.15f;                          // Decrease The Rot
return TRUE;                            // 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.

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

HDC          hDC=NULL; // Privat
HGLRC        hRC=NULL; // Perman
HWND         hWnd=NULL;
HINSTANCE hInstance; // Holds '

bool         keys[256];
bool         active=TRUE;
bool         fullscreen=TRUE; // Fullsc:

GLfloat      xrot;
GLfloat      yrot;
GLfloat      zrot;

GLuint       texture[1];

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

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 i
{
    FILE *File=NULL; // File H
```

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"); // Check f
```

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 T
    {
        fclose(File);
        return auxDIBImageLoad(Filename); // Load T
    }
```

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()
{
```

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; // Status
```

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]; // Create
```

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

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

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 won't 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]);
```

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.

```
if (TextureImage[0])
{
    if (TextureImage[0]->data) // If Tex
    {
        free(TextureImage[0]->data);
    }

    free(TextureImage[0]);
}
```

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;
}
```

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;
    }

    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
    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]); // Select
```

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 right 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, 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
    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 Ri
    // Bottom Face
    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
    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 Ri
    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(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();
```

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 too 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 Herboth](#) )
- \* [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.

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

HDC          hDC=NULL; // Private
HGLRC        hRC=NULL; // Permanent
HWND         hWnd=NULL;
HINSTANCE hInstance; // Holds 'Instance' of the program

bool         keys[256];
bool         active=TRUE;
bool         fullscreen=TRUE; // Fullscreen
```

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 **lp** 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; // Depth
```

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 }; // Ambient
```

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 }; // Diffuse
```

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.

```
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f }; // Light
```

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); // Declar
```

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 B
{
    FILE *File=NULL; // File H

    if (!Filename)
    {
        return NULL;
    }

    File=fopen(Filename,"r"); // Check '

    if (File) // Does T
    {
        fclose(File);
        return auxDIBImageLoad(Filename); // Load T
    }
    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]; // Create
memset(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, a lot 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 (TextureImage[0])
{
    if (TextureImage[0]->data) // If Tex
    {
        free(TextureImage[0]->data);
    }
    free(TextureImage[0]);
}
```

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 won't 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)
{
    glClearColor(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);                                           // Transl.
    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);                                                  // Start '

```

`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,

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 1
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Point 2
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Point 3
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point 4
// Back Face
glNormal3f( 0.0f, 0.0f,-1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point 1
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point 2
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Point 3
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Point 4
// Top Face
glNormal3f( 0.0f, 1.0f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point 1
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point 2
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Point 3
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Point 4
// Bottom Face
glNormal3f( 0.0f,-1.0f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point 1
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Point 2
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Point 3
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point 4
// Right face
glNormal3f( 1.0f, 0.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Point 1
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Point 2
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Point 3
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Point 4
// Left Face
glNormal3f(-1.0f, 0.0f, 0.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point 1
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point 2
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point 3
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point 4
glEnd(); // Done Drawing

```

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()`. We're 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 `lp` isn't false, meaning 'L' has already been pressed once or it's being held down, nothing will happen.

```

SwapBuffers(hdc); // Swap B
if (keys['L'] && !lp)
{

```

If **lp** was false, meaning the 'L' key hasn't been pressed yet, or it's been released, **lp** 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 **lp** 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 or 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; // lp Bec
light=!light;

```

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 **lp** 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.

```

if (!keys['L'])
{
    lp=FALSE; // If So,
}

```

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_UP])        // Is Up
{
    xspeed-=0.01f;
}
if (keys[VK_DOWN])     // Is Dow
{
    xspeed+=0.01f;
}

```

```

    }
    if (keys[VK_RIGHT])
    {
        yspeed+=0.01f;
    }
    if (keys[VK_LEFT]) // Is Left
    {
        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 :
        {
            keys[VK_F1]=FALSE; // If So I
            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.

$$(R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s S_a + A_d D_a)$$

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 (A<sub>s</sub>, A<sub>s</sub>, A<sub>s</sub>, A<sub>s</sub>) (AKA source alpha) for S and (1, 1, 1, 1) - (A<sub>s</sub>, A<sub>s</sub>, A<sub>s</sub>, A<sub>s</sub>) (AKA one minus src alpha) for D. This will yield a blending equation that looks like this:

$$(R_s A_s + R_d (1 - A_s), G_s A_s + G_d (1 - A_s), B_s A_s + B_d (1 - A_s), A_s A_s + A_d (1 - A_s))$$

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 coming 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
#include <stdio.h> // Header File For Standard Input/Output
#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 GLaux Library

HDC hDC=NULL; // Private GDI Device Context
HGLRC hRC=NULL; // Permanent Rendering Context
HWND hWnd=NULL; // Holds Our Window Handle
HINSTANCE hInstance; // Holds The Instance Of The Application

bool keys[256]; // Array Used For The Keyboard State
bool active=TRUE; // Window Active Flag Set To TRUE
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen
bool light; // 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.

```
texture1 = 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% Alpha
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 I
{
    bp=TRUE;                       // If So, bp Becomes TRUE
    blend = !blend;                // Toggle blend TRUE / FALSE
    if(blend)                      // Is blend TRUE?
    {
        glEnable(GL_BLEND);        // Turn Blending On
        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](mailto: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 Herboth](#) )
- \* 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> // Header File For
#include <stdio.h> // Header File For Standard
#include <gl\gl.h> // Header File For The OpenC
#include <gl\glu.h> // Header File For
#include <gl\glaux.h> // Header File For

HDC hDC=NULL; // Private GDI Device Context
HGLRC hRC=NULL; // Permanent Rendering Context
HWND hWnd=NULL; // Holds Our Window
HINSTANCE hInstance; // Holds The Instance Of The

bool keys[256]; // Array Used For Key State
bool active=TRUE; // Window Active Flag
bool fullscreen=TRUE; // Fullscreen Flag Set To Full
```

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, released **tp**=FALSE).

```
BOOL twinkle; // Twinkling Stars
BOOL tp; // '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**.

```
typedef struct                                // Create A Structu
{
    int r, g, b;                               // Stars Color
    GLfloat dist;                             // Stars Distance I
    GLfloat angle;                            // Stars Current An
}
stars;                                        // Structures Name
stars star[num];                             // 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 Va
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 NULL
    }

    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 Return
    }
    return NULL; // If Load Failed !
}
```

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

    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 To TRUE

        glGenTextures(1, &texture[0]); // Create One Texture

        // 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]->sizeY, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
    }

    if (TextureImage[0]) // If Texture Exists
    {
        if (TextureImage[0]->data) // If Texture Image Exists
        {
            free(TextureImage[0]->data); // Free The Texture Image Data
        }

        free(TextureImage[0]); // Free The Image
    }
}
```

```

    }

    return Status; // Return The Status
}

```

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't Load
    }

    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 Setup
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective
    glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Set The Blending Function to GL_BLEND
    glEnable(GL_BLEND); // 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!

```

for (loop=0; loop<num; loop++) // Create A Loop Through All Stars
{
    star[loop].angle=0.0f; // Start All The Stars At 0.0f
}

```

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 a lot 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 Distance
star[loop].r=rand()%256; // Give star[loop] A Random Color
star[loop].g=rand()%256; // Give star[loop] A Random Color
star[loop].b=rand()%256; // Give star[loop] A Random Color
}

```

```

        return TRUE; // Initialization
    }

```

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 All
    {
        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 (1

```

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-loop)-1].b,255);
    glBegin(GL_QUADS); // Begin Drawing The Texture
        glVertex3f(-1.0f,-1.0f, 0.0f);
        glVertex3f( 1.0f,-1.0f, 0.0f);
        glVertex3f( 1.0f, 1.0f, 0.0f);
        glVertex3f(-1.0f, 1.0f, 0.0f);
    glEnd(); // 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
    glVertex3f(-1.0f,-1.0f, 0.0f);
    glVertex3f( 1.0f,-1.0f, 0.0f);
    glVertex3f( 1.0f, 1.0f, 0.0f);
    glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd(); // Done Drawing The Texture

```

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; // Used To Spin The
star[loop].angle+=float(loop)/num; // Changes The Angle Of A Star
star[loop].dist-=0.01f; // Changes The Distance

```

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
}

```

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 Buff
    if (keys['T'] && !tp)                        // Is T Being Press
    {
        tp=TRUE;                                // If So, Make tp TRUE
        twinkle=!twinkle;                       // Make twinkle Equal The Op
    }

```

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 Pressed
    {
        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 Cr
        }
    }
}

```

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 of your 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 Herboth](#) )
- \* [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 categorize 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 Triangles
    TRIANGLE* triangle;                                    // Pointer To Array Of Triangles
} SECTOR;                                                  // Call It SECTOR
```

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

```
typedef struct tagTRIANGLE                                  // Build Our Triangle
{
    VERTEX vertex[3];                                       // Array Of Three Vertices
} TRIANGLE;                                               // Call It TRIANGLE
```

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

```
typedef struct tagVERTEX                                    // Build Our Vertex
{
    float x, y, z;                                          // 3D Coordinates
    float u, v;                                             // Texture Coordinates
} VERTEX;                                                  // Call It 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:

```
void readstr(FILE *f, char *string)                   // Read In A String
{
    do return;                                       // Start
    {
        fgets(string, 255, f);                       // Read O
    } while ((string[0] == '/') || (string[0] == '\n')); // See If It Is Wo
    return;                                          // Jump B
}
```

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:

```
NUMPOLLIIES n
```

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

```
int numtriangles;                                     // Number Of Triang
```

```

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 Triangles
float x, y, z, u, v; // 3D And
...
sector1.triangle = new TRIANGLE[numtriangles]; // Allocate
sector1.numtriangles = numtriangles; // Define The Number
// Step Through Each Triangle In Sector
for (int triloop = 0; triloop < numtriangles; triloop++) // Loop Through All
{
    // Step Through Each Vertex In Triangle
    for (int vertloop = 0; vertloop < 3; vertloop++) // Loop Through All
    {
        readstr(filein,oneline); // Read String To Store
        // Read Data Into Respective Vertex Values
        sscanf(oneline, "%f %f %f %f %f %f", &x, &y, &z, &u, &v);
        // Store Values Into Respective Vertices
        sector1.triangle[triloop].vertex[vertloop].x = x; // Sector 1, Triangle
        sector1.triangle[triloop].vertex[vertloop].y = y; // Sector 1, Triangle
        sector1.triangle[triloop].vertex[vertloop].z = z; // Sector 1, Triangle
        sector1.triangle[triloop].vertex[vertloop].u = u; // Sector 1, Triangle
        sector1.triangle[triloop].vertex[vertloop].v = v; // Sector 1, Triangle
    }
}
}

```

Each triangle in our data file is declared as follows:

```

x1 y1 z1 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
}

if (keys[VK_LEFT]) // Is The Left Arrow
{
    yrot += 1.5f; // Rotate
}

if (keys[VK_UP]) // Is The Up Arrow
{
    xpos -= (float)sin(yrot*piover180) * 0.05f; // Move On The X-Plane
    zpos -= (float)cos(yrot*piover180) * 0.05f; // Move On The Z-Plane
    if (walkbiasangle >= 359.0f) // Is walkbiasangle
    {
        walkbiasangle = 0.0f; // Make walkbiasangle
    }
    else // Otherwise
    {
        walkbiasangle+= 10; // If walkbiasangle
    }
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f; // Causes
}

if (keys[VK_DOWN]) // Is The Down Arrow
{
    xpos += (float)sin(yrot*piover180) * 0.05f; // Move On The X-Plane
    zpos += (float)cos(yrot*piover180) * 0.05f; // Move On The Z-Plane
    if (walkbiasangle <= 1.0f) // Is walkbiasangle
    {
        walkbiasangle = 359.0f; // Make walkbiasangle
    }
    else // Otherwise
    {
        walkbiasangle-= 10; // If walkbiasangle
    }
    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 appropriately. 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 separate function.

```

int DrawGLScene(GLvoid) // Draw The Scene
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And
    glLoadIdentity(); // Reset The Current
}

```

```

GLfloat x_m, y_m, z_m, u_m, v_m; // Floating Point
GLfloat xtrans = -xpos; // Used For
GLfloat ztrans = -zpos; // Used For
GLfloat ytrans = -walkbias-0.25f; // Used For Bounci
GLfloat sceneroty = 360.0f - yrot; // 360 Degree Angl

int numtriangles; // Integer To Hold

glRotatef(lookupdown,1.0f,0,0); // Rotate
glRotatef(sceneroty,0,1.0f,0); // Rotate

glTranslatef(xtrans, ytrans, ztrans); // Transl.
glBindTexture(GL_TEXTURE_2D, texture[filter]); // Select

numtriangles = sector1.numtriangles; // Get The Number (

// Process Each Triangle
for (int loop_m = 0; loop_m < numtriangles; loop_m++) // Loop Through All
{
    glBegin(GL_TRIANGLES); // Start
        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
    glEnd(); // Done Drawing Tr
}
return TRUE; // Jump B
}

```

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 ([Betelgeuse](#))

- \* [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 quads x 44 quads. **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]; // The Array For Tl
int wiggle_count = 0; // Counter Used To
GLfloat hold; // 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.

```
glPolygonMode( GL_BACK, GL_FILL ); // Back Face Is Filled In
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(
    }
}
```

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 Scene
{
    int x, y; // Loop Variables
    float float_x, float_y, float_xb, float_yb; // Used To Break The Flag Ir
```

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 Depth
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.

```

glBegin(GL_QUADS);                       // Start Drawing Our Quads
for( x = 0; x < 44; x++ )                // Loop Through The X Plane
{
    for( y = 0; y < 44; y++ )            // Loop Through The Y Plane
    {

```

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 X
float_y = float(y)/44.0f;                 // Create A Floating Point Y
float_xb = float(x+1)/44.0f;              // Create A Floating Point X
float_yb = float(y+1)/44.0f;              // Create A Floating Point Y

```

We use the four variables above for the texture coordinates. Each of our polygons (square in the grid), has a 1/44 x 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][2] );

glTexCoord2f( float_xb, float_yb );      // Third Texture Coordinate
glVertex3f( points[x+1][y+1][0], points[x+1][y+1][1], points[x+1][y+1][2] );

glTexCoord2f( float_xb, float_y );       // Fourth Texture Coordinate
glVertex3f( points[x+1][y][0], points[x+1][y][1], points[x+1][y][2] );
    }
}
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 Down
{

```

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 Value
    for( x = 0; x < 44; x++ ) // 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 Becomes First
}
wiggle_count = 0; // Set Counter Back To Zero
}
wiggle_count++; // Increase The Counter

```

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 Rotation
yrot+=0.2f; // Increase The Y Rotation
zrot+=0.4f; // Increase The Z Rotation

return 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](mailto: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;
HINSTANCE hInstance;

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

```
// Header File For Windows
// Header File For Standard Input/Out
// Header File For The OpenGL32 Libræ
// Header File For The GLu32
// Header File For The GLaux

// Private GDI Device Context
// Permanent Rendering Context
// Holds Our Window Handle
// Holds The Instance Of The Applicat

// Array Used For The Keyboæ
// Window Active Flag Set To
// 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 texture[1]; // 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 Axis
GLfloat yrot; // Rotates Cube On The Y Axis
```

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.

```
static GLfloat boxcol[5][3]= // Array For Box Colors
{
    // Bright: Red, Orange, Yellow, Green, Blue
    {1.0f,0.0f,0.0f},{1.0f,0.5f,0.0f},{1.0f,1.0f,0.0f},{0.0f,1.0f,0.0f},{0.0f,1.0f,1.0f};
};

static GLfloat topcol[5][3]= // Array For Top Colors
{
    // Dark: Red, Orange, Yellow, Green, Blue
    {.5f,0.0f,0.0f},{0.5f,0.25f,0.0f},{0.5f,0.5f,0.0f},{0.0f,0.5f,0.0f},{0.0f,0.5f,0.5f};
};

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

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 every 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 won't 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);
    // Bottom Face
    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);
    // Front 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);
    // Back Face
    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);
    // Right face
    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
        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 our 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.

```

glBegin(GL_QUADS);
    // Top Face
    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);
glEnd();

```

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
    {
        return FALSE; // If Texture Didn
    }
    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 Dep
```

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); // Quick And Dirty
glEnable(GL_LIGHTING); // Enable Lighting
glEnable(GL_COLOR_MATERIAL); // 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++)                          // 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<yloop;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 confusing, 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 C
```

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:
{
    yrot-=0.2f;                          // If So Spin Cubes
}
if (keys[VK_RIGHT])                     // Right Arrow Bei
{
    yrot+=0.2f;                          // If So Spin Cube:
}
if (keys[VK_UP])                        // Up Arrow Being Pressed?
{
    xrot-=0.2f;                          // If So Tilt Cube:
}
if (keys[VK_DOWN])                     // Down Arrow Being Pressed:
{
    xrot+=0.2f;                          // If So Tilt Cube:
}

```

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
    {
        return 0;                       // Quit If Window Was Not Cr
    }
}
}
}

```

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)**

- \* [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 independent 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 GLaux Library

HDC          hDC=NULL; // Private GDI Device Context
HGLRC        hRC=NULL; // Permanent Rendering Context
HWND         hWnd=NULL; // Holds Our Window Handle
HINSTANCE hInstance; // 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

    base = glGenLists(96);               // 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.

```
0,                                        // Width Of Font
```

Angle of Escapement will rotate the font. Unfortunately this isn't a very useful feature. Unless your angle is at 0, 90, 180, and 270 degrees, the font usually gets cropped to fit inside its 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 :(

```
0, // Angle Of Escapement
0, // Orientation Angle
```

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 a lot 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 won't be. Pretty simple :)

```
FALSE, // Italic
FALSE, // 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 Identifier
```

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 tries 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 Preci:
```

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 Creat
wglUseFontBitmaps(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)           // Delete The Font
{
    glDeleteLists(base, 96);       // 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`.

```
GLvoid glPrint(const char *fmt, ...) // Custom GL "Print" Routine
{
```

The first line below creates storage space for a 256 character string. `text` is the string we will end up printing to the screen. The second line below 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 will point to them.

```
    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) // If There's No Text
        return;     // 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); // Parses The String For Var
    vsprintf(text, fmt, ap); // And Converts Symbols
    va_end(ap); // Results Are Stored In Text
```

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); // Pushes The Display List Flag
    glListBase(base - 32); // Sets The Base Call
```

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 we're 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 each other. 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 Text
        glPopAttrib();                                        // Pops The Display List Attributes
    }

```

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 Here
{
    glShadeModel(GL_SMOOTH);                             // Enable Smooth Shading
    glClearColor(0.0f, 0.0f, 0.0f, 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 Camera

    BuildFont();                                         // Build The Font

    return TRUE;                                         // Initialization Succeeded
}

```

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 won't 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 translate 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+cnt2)));
```

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. As an example, if `cnt1` was equal to `300.12345f` the number we would end up seeing 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", cnt1); // Print GL Text To The Screen
```

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

```

cnt1+=0.051f; // Increase The First Counter
cnt2+=0.005f; // Increase The Second Counter
return TRUE; // Everything Went Well
}

```

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 Class
{
    MessageBox(NULL,"Could Not Unregister Class. ","SHUTDOWN ERROR",MB_OK | MB_ICONERROR);
    hInstance=NULL; // Set hInstance To NULL
}

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.

```
#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 GLaux Library

HDC          hDC=NULL; // Private GDI Device Context
HGLRC        hRC=NULL; // Permanent Rendering Context
HWND         hWnd=NULL; // Holds Our Window Handle
HINSTANCE hInstance; // 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

    base = glGenLists(256); // 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.

```
0, // 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
0, // 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, // Italic
FALSE, // 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 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 tries 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 Precisi
```

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 '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 Current
                      0, // Starting Character
                      255, // Number Of Display Lists
                      base, // Starting Display List Value
```

That's not all however. We then set the deviation level. The closer to 0.0f, the smoother 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 Flat
    0.2f, // Font Thickness :
    WGL_FONT_POLYGONS, // Use Polygons, Not Lines
    gmf); // Address Of Buffer
}
```

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)                                   // If There's No Text
        return;                                       // 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);                                // Parses The String For Var
    vsprintf(text, fmt, ap);                           // And Converts Sy
    va_end(ap);                                       // Results Are Sto
```

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.

```
for (unsigned int loop=0;loop<(strlen(text));loop++) // Loop To Find Text Length
{
    length+=gmf[text[loop]].gmfCellIncX;           // Increase Length By Each C
}
```

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); // Pushes The Display List I
glListBase(base); // 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 each other. 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 To
        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 quick 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
{
    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 Depth
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);   // Really Nice Perspective (
    glEnable(GL_LIGHT0);                                 // Enable Default Light
    glEnable(GL_LIGHTING);                               // Enable Lighting
    glEnable(GL_COLOR_MATERIAL);                         // Enable Coloring

    BuildFont();                                        // Build The Font

    return TRUE;                                        // Initialization Done
}

```

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); // Move Ten Units :
```

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 than 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; // Increase The Rot
return TRUE; // 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 To
}
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.

```
#include <windows.h> // Header File For Windows
#include <math.h> // Header File For Windows Math Libra
#include <stdio.h> // Header File For Standard Input/Out
#include <gl\gl.h> // Header File For The OpenGL32 Libra
#include <gl\glu.h> // Header File For The GLu32
#include <gl\glaux.h> // Header File For The GLaux

HDC hDC=NULL; // Private GDI Device Context
HGLRC hRC=NULL; // Permanent Rendering Context
HWND hWnd=NULL; // Holds Our Window Handle
HINSTANCE hInstance; // Holds The Instance Of The Applicat

bool keys[256]; // Array Used For The Keyboar
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[256];                          // Address Buffer For Font Storage
    HFONT font;                                          // Windows Font ID

    base = glGenLists(256);                               // Storage For 256
    font = CreateFont(-12,                               // Height Of Font
                     0,                                  // Width Of Font
                     0,                                  // Angle Of Escaper
                     0,                                  // 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 Precision
    CLIP_DEFAULT_PRECIS,                              // Clipping Precision
    ANTIALIASED_QUALITY,                              // Output Quality
    FF_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 ( Modified

```

The remaining lines of code have not changed.

```
SelectObject(hdc, font); // Selects The Font We Create

wglUseFontOutlines(      hdc, // Select The Current
                        0,    // Starting Character
                        255,  // Number Of Displayable
                        base, // Starting Displayable
```

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 Outline
```

The next three lines of code are still the same.

```
0.2f, // Font Thickness :
WGL_FONT_POLYGONS, // Use Polygons, Not Lines
gmf); // Address Of Buffer
}
```

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
{
    FILE *File=NULL; // File Handle

    if (!Filename) // Make Sure A Filename
    {
        return NULL; // If Not Return NULL
    }

    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 Return
    }

    return NULL; // If Load Failed I
```

```

}

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 To TRUE

        glGenTextures(1, &texture[0]); // Create The Texture

        // Build Linear Mipmapped Texture
        glBindTexture(GL_TEXTURE_2D, texture[0]);
        gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY,
        GL_RGB, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
    }
}

```

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 its 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` - Everyone's favorite. Creates a metallic reflective type object.

It's important to note that I'm leaving out a lot 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 Status
}

```

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
{
    if (!LoadGLTextures())                // Jump To Texture Loading
    {
        return FALSE;                    // If Texture Didn't Load
    }
    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
    glEnable(GL_DEPTH_TEST);              // Enables Depth Testing
    glDepthFunc(GL_LEQUAL);                // The Type Of Depth Testing
    glEnable(GL_LIGHT0);                   // Quick And Dirty Lighting
    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 Complete
}

```

The resize code hasn't changed, but our DrawGLScene code has.

```

int DrawGLScene(GLvoid)                   // Here's Where We Draw
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
    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); // Rotate On The X
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
```

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"); // Draw A Skull And
rot+=0.1f; // Increase The Rot
return TRUE; // 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
}

KillFont(); // Destroy The Font
}
```

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;                                     // G Pressed? ( New )
GLuint  filter;                                // Which Filter To Use
GLuint  fogMode[]= { GL_EXP, GL_EXP2, GL_LINEAR }; // Storage For Three Types Of Fog
GLuint  fogfilter= 0;                          // Which Fog To Use
GLfloat fogColor[4]= {0.5f, 0.5f, 0.5f, 1.0f}; // Fog Color
```

### DrawGLScene Setup

Now that we have established our variables we will move down to InitGL. The glColor() 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);           // We'll Clear To The Color Of The Fog
glFogf(GL_FOG_MODE, fogMode[fogfilter]);    // Fog Mode
glFogfv(GL_FOG_COLOR, fogColor);           // Set Fog Color
glFogf(GL_FOG_DENSITY, 0.35f);              // How Dense Will The Fog Be
glHint(GL_FOG_HINT, GL_DONT_CARE);         // Fog Hint Value
glFogf(GL_FOG_START, 1.0f);                 // Fog Start Depth
glFogf(GL_FOG_END, 5.0f);                  // Fog End Depth
glEnable(GL_FOG);                           // 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 or 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'] && !gp) // Is The G Key Being Pressed
{
    gp=TRUE; // gp Is Set To TRUE
    fogfilter++; // Increase fogfilter By One
    if (fogfilter>2) // Is fogfilter Greater Than 2?
    {
        fogfilter=0; // If So, Set fogfilter To 0
    }
    glFogi (GL_FOG_MODE, fogMode[fogfilter]); // Fog Mode
}
if (!keys['G']) // Has The G Key Been Released
{
    gp=FALSE; // If So, gp 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](mailto: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 everyone's 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 phrase 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 requires a lot 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... Let's 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 GLaux Library

HDC          hDC=NULL; // Private GDI Device Context
HGLRC        hRC=NULL; // Permanent Rendering Context
HWND         hWnd=NULL; // Holds Our Window Handle
HINSTANCE hInstance; // Holds The Instance Of The Application

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
```

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   base;                // Base Display List For The Font
GLuint   texture[2];         // Storage For Our Font Texture
GLuint   loop;               // Generic Loop Variable

GLfloat  cnt1;               // 1st Counter Used To Move Text & For Coloring
GLfloat  cnt2;               // 2nd Counter Used To Move Text & For Coloring

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

```

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 .
{
    FILE *File=NULL;                                           // File H
    if (!Filename)
    {
        return NULL;
    }
    File=fopen(Filename,"r");                                   // Check '
    if (File)                                                 // Does T
    {
        fclose(File);
        return auxDIBImageLoad(Filename);                       // Load T
    }
    return NULL;
}

```

The following 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;                                         // Status
    AUX_RGBImageRec *TextureImage[2];                         // 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.

```

    memset(TextureImage,0,sizeof(void *)*2);                 // Set Th
    if ((TextureImage[0]=LoadBMP("Data/Font.bmp")) &&
        (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 your 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.

```

        glGenTextures(2, &texture[0]);

        for (loop=0; loop<2; loop++)
        {
            // Build All The Textures
            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
        }
    }

```

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.

```

        for (loop=0; loop<2; loop++)
        {
            if (TextureImage[loop])
            {
                if (TextureImage[loop]->data)
                {
                    free(TextureImage[loop]->data);
                }
                free(TextureImage[loop]); // Free T
            }
        }
        return Status;
    }
}

```

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 t
{

```

The following two variables 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 its own display lists.

```
for (loop=0; loop<256; loop++)
{
```

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 remainder 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 give 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 remainder 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;           // X Posi
cy=float(loop/16)/16.0f;         // 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.

```
glNewList(base+loop, GL_COMPILE); // Start :
```

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 each other. 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: A lot can be added to this code, such as variable support, character sizing, spacing, and a lot 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. If **set** is 1 or greater the second 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]); // Select
```

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); // Disabl.
```

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(); // Reset '
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); // Select
glPushMatrix();
glLoadIdentity(); // Reset '
```

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.

```
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(string)` 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); // Write '
```

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); // Select
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); // Enable.
}
```

Nothing has changed in `ReSizeGLScene()` so we'll skip right to `InitGL()`.

```
int InitGL(GLvoid) // All Se
{
```

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.

```
if (!LoadGLTextures())
{
    return FALSE;
}
```

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); // Enable
glDepthFunc(GL_LEQUAL);
glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Select
glShadeModel(GL_SMOOTH); // Enable
glEnable(GL_TEXTURE_2D); // Enable
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);
```

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); // Bright
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

```

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.

```

glRotatef(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 0

glColor3f(1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt2)),1.0f*float(cos(cnt1+cnt2)));
glPrint(int((280+230*cos(cnt2)),int(235+200*sin(cnt1)),"OpenGL",1); // Print 1
```

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 Color to Dark Blue
glPrint(int(240+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0); // Draw Text

glColor3f(1.0f,1.0f,1.0f); // Set Color to White
glPrint(int(242+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0); // Draw Text
```

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(
    HINSTANCE hInstance, // Instance
    HINSTANCE hPrevInstance, // Previous Instance
    LPSTR lpCmdLine, // Command Line
    int nCmdShow) // Window Show Command

{
    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 Mode
    }
}
```

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 To
}

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

HDC          hDC=NULL;
HGLRC        hRC=NULL;
HWND         hWnd=NULL;
HINSTANCE hInstance;

bool keys[256];
bool active=TRUE;
bool fullscreen=TRUE;
bool light;
bool lp;
bool fp;
bool sp;

int part1;
int part2;
int p1=0;
int p2=1;

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

GLfloat z=-5.0f;

GLUquadricObj *quadratic;

GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f };

GLuint filter;
GLuint texture[3];
GLuint object=0;

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

```

```

// Header File For Windows
// Header File For Standard Input/Output
// Header File For The OpenGL32 Library
// Header File For The GLU32 Library
// Header File For The GLaux Library

// Private GDI Device Context
// Permanent Rendering Context
// Holds Our Window Handle
// Holds The Instance Of The Application

// Array Used For The Keyboard
// Window Active Flag Set To TRUE
// Fullscreen Flag Set To Fullscreen
// Lighting ON/OFF
// L Pressed?
// F Pressed?
// Spacebar Pressed?

// Start Of Disc ( NEW )
// End Of Disc
// Increase 1
// Increase 2

// X Rotation
// Y Rotation
// X Rotation Speed
// Y Rotation Speed

// Depth Into The Screen

// Storage For Our Quadratic Objects

// Ambient Light Values
// Diffuse Light Values
// Light Position

// Which Filter To Use
// Storage for 3 textures
// Which Object To Draw ( NEW )

```

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();           // Create A Pointer To The Quadric Ok
gluQuadricNormals(quadratic, GLU_SMOOTH); // Create Smooth Normals ( NEW )
gluQuadricTexture(quadratic, GL_TRUE); // Create Texture Coords ( 1

```

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 Ri
    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 Ri
    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
    glNormal3f( 0.0f, 1.0f, 0.0f); // Normal Facing Up
    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 Ri
    // 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 Ri
    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 Ri
    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
{
    glClearColor(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 Screen

    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 :
        glDrawCube(); // Draw Our Cube
        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 fourth 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 thin 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 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 slower it will run.

```

case 2:
    gluDisk(quadratic,0.5f,1.5f,32,32); // Drawing Object :
    break; // Draw A Disc (CD Shape)
           // 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:
    gluSphere(quadratic,1.3f,32,32); // Drawing Object :
    break; // Draw A Sphere
           // 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:
    glTranslatef(0.0f,0.0f,-1.5f); // Drawing Object !
    gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32); // Center The Cone
    break; // A Cone With A Bottom Radi
           // Done

```

Our sixth object is created with gluPartialDisk. 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:
    part1+=p1; // Drawing Object !
    part2+=p2; // Increase Start i
               // Increase Sweep i

    if(part1>359) // 360 Degrees
    {
        p1=0; // Stop Increasing
        part1=0; // Set Start Angle To Zero
        p2=1; // Start Increasing
        part2=0; // Start Sweep Angle At Zero
    }
    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

```

```
};  
  
xrot+=xspeed;           // Increase Rotatio  
yrot+=yspeed;          // Increase Rotatio  
return TRUE;           // Keep Going  
}
```

Now for the final part, the key input. Just add this where we check the rest of key input.

```
if (keys[' '] && !sp)           // Is Spacebar Bei  
{  
    sp=TRUE;                   // If So, Set sp To TRUE  
    object++;                  // Cycle Through The Objects  
    if(object>5)               // Is object Great  
        object=0; // If So, Set To Zero  
}  
if (!keys[' '])                // Has The Spacebar  
{  
    sp=FALSE;                  // If So, Set sp To FALSE  
}
```

That's 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](http://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 snippets 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>                // Header File For Windows
#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 GLaux Library

#define MAX_PARTICLES    1000       // Number Of Particles To Create ( NEW )

HDC          hdc=NULL;             // Private GDI Device Context
```

```

HGLRC          hRC=NULL;          // Permanent Rendering Context
HWND           hWnd=NULL;        // Holds Our Window Handle
HINSTANCE hInstance;             // Holds The Instance Of The Application

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
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    slowdown=2.0f;           // 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    texture[1];             // 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 **r**, **g** and **b** hold the red intensity, green intensity and blue intensity of our particle. The closer **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;                                       // Red Value
float    g;                                       // Green Value
float    b;                                       // Blue Value
```

The variables **x**, **y** and **z** control where the particle will be displayed on the screen. **x** holds the location of our particle on the x axis. **y** holds the location of our particle on the y axis, and finally **z** holds the location of our particle on the z axis.

```
float    x;                                       // X Position
float    y;                                       // Y Position
float    z;                                       // 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 positive 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.

```

float    xg;           // X Gravity
float    yg;           // Y Gravity
float    zg;           // Z Gravity

```

**particles** is the name of our structure.

```

}
particles;           // Particles Structure

```

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.

```

static GLfloat colors[12][3]=               // Rainbow Of Colors
{
    {1.0f,0.5f,0.5f},{1.0f,0.75f,0.5f},{1.0f,1.0f,0.5f},{0.75f,1.0f,0.5f},
    {0.5f,1.0f,0.5f},{0.5f,1.0f,0.75f},{0.5f,1.0f,1.0f},{0.5f,0.75f,1.0f},
    {0.5f,0.5f,1.0f},{0.75f,0.5f,1.0f},{1.0f,0.5f,1.0f},{1.0f,0.5f,0.75f}
};

```

```

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

```

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
    {
        return NULL;                               // If Not 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 texture. 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

    memset(TextureImage,0,sizeof(void *)*1); // 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 To TRUE
    glGenTextures(1, &texture[0]); // Create One 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]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
}

if (TextureImage[0]) // If Texture Exists
{
    if (TextureImage[0]->data) // If Texture Image Exists
    {
        free(TextureImage[0]->data); // Free The Texture Image Data
    }
    free(TextureImage[0]); // Free The Image Object
}
return Status; // Return The Status
}
```

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 By Zero Error
    {
        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

    // Calculate The Aspect Ratio Of The Window
    gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 200.0f); // ( MODIFIED )
}
```

```

    glMatrixMode(GL_MODELVIEW);           // Select The Modelview Matrix
    glLoadIdentity();                   // Reset The Modelview Matrix
}

```

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.

```

int InitGL(GLvoid)                       // All Set
{
    if (!LoadGLTextures())
    {
        return FALSE;
    }
}

```

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 Smooth Shading
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);    // Black Background
glClearDepth(1.0f);                      // Enable Depth Testing
glEnable(GL_DEPTH_TEST);                 // Enable Depth Testing
glEnable(GL_BLEND);                      // Enable Blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE);       // Type Of Blending: Source Alpha, Dest One
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Corrections
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST); // Really Nice Points
glEnable(GL_TEXTURE_2D);                 // Enable 2D Texturing
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Particle Texture

```

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.

```

for (loop=0; loop<MAX_PARTICLES; loop++)
{
    particle[loop].active=true;           // Make A Particle Active
    particle[loop].life=1.0f;             // Give A Particle Full Life
}

```

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 Fade Speed

```

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.

```
particle[loop].r=colors[(loop+1)/(MAX_PARTICLES/12)][0]; // Select
particle[loop].g=colors[(loop+1)/(MAX_PARTICLES/12)][1]; // Select
particle[loop].b=colors[(loop+1)/(MAX_PARTICLES/12)][2]; // 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.

```
particle[loop].xi=float((rand()%50)-26.0f)*10.0f; // Random
particle[loop].yi=float((rand()%50)-25.0f)*10.0f; // Random
particle[loop].zi=float((rand()%50)-25.0f)*10.0f; // 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; // Set Ve:
particle[loop].zg=0.0f;
}
return TRUE;
}
```

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 translations, that way we don't alter the modelview matrix while drawing our particles.

```
int DrawGLScene(GLvoid)
{
```

```

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);           // Clear
glLoadIdentity();                                             // Reset

```

---

We start off by creating a loop. This loop will update each one of our particles.

---

```

for (loop=0;loop<MAX_PARTICLES;loop++)
{

```

First thing we do is check to see if the particle is active. If it's not active, it won't 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;                             // Grab O
        float y=particle[loop].y;                             // Grab O
        float z=particle[loop].z+zoom;

```

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 1.0. 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

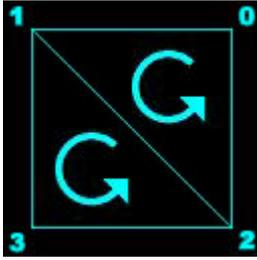
```

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 a lot 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].x+=particle[loop].xi/(slowdown*1000); // Move O
```

```
particle[loop].y+=particle[loop].yi/(slowdown*1000); // Move O:
particle[loop].z+=particle[loop].zi/(slowdown*1000); // Move O:
```

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)
{
```

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]
```

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]
```

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.

```

        if (keys[VK_TAB])
        {
            particle[loop].x=0.0f;
            particle[loop].y=0.0f;
            particle[loop].z=0.0f;
            particle[loop].xi=float((rand()%50)-26.0f)*10.0f;
            particle[loop].yi=float((rand()%50)-25.0f)*10.0f;
            particle[loop].zi=float((rand()%50)-25.0f)*10.0f;
        }
    }
}
return TRUE;
}

```

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.

```

int WINAPI WinMain(
    HINSTANCE Instance,
    HINSTANCE PrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow)
{
    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; // Windowed Mode
    }
}

```

```

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

```

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 (msg.message==WM_QUIT) // Have We Receive
        {
            done=TRUE; // If So
        }
        else // If Not
        {
            TranslateMessage(&msg); // Transl.
            DispatchMessage(&msg); // Dispat
        }
    }
    else // If The
    {
        if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Updati
        {
            done=TRUE; // ESC or
        }
        else // Not Ti
        {
            SwapBuffers(hdc); // Swap Buffers (D

```

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 In
```

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 Out
```

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 Pressed
{
    rp=true; // Set Flag Telling
    rainbow=!rainbow; // Toggle Rainbow Mode
}
if (!keys[VK_RETURN]) rp=false; // If Released
```

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 Telling
delay=0; // Reset The Rainb
col++; // 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 1
```

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;
```

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;
```

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 Co
```

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 Particle Tutorial"
    {
        return 0; // Quit If Window Was Not Cr
    }
}
}
}
// Shutdown
KillGLWindow(); // Kill The Window
return (msg.wParam); // Exit The Program
}
```

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!

```
#include <windows.h> // Header File For Windows
#include <math.h> // Header File For Windows Math Libra
#include <stdio.h> // Header File For Standard Input/Out
#include <gl\gl.h> // Header File For The OpenGL32 Libra
#include <gl\glu.h> // Header File For The GLu32
#include <gl\glaux.h> // Header File For The Glau

HDC hDC=NULL; // Private GDI Device Context
HGLRC hRC=NULL; // Permanent Rendering Context
HWND hWnd=NULL; // Holds Our Window Handle
HINSTANCE hInstance; // 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 or 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]; // Array Used For The Keyboa
bool active=TRUE; // Window Active Flag Set To
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen
bool masking=TRUE; // Masking On/Off
```

```

bool    mp;                                // M Pressed?
bool    sp;                                // Space Pressed?
bool    scene;                             // Which Scene To Draw

GLuint   texture[5];                       // Storage For Our Five Text
GLuint   loop;                             // Generic Loop Variable

GLfloat  roll;                             // Rolling Texture

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Declaration 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]**.

```

int LoadGLTextures()                                // Load B
{
    int Status=FALSE;                               // Status Indicator
    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 Th
        glGenTextures(5, &texture[0]);                    // Create

        for (loop=0; loop<5; loop++)                      // 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++)                          // Loop T
    {
        if (TextureImage[loop])                            // If Tex
        {
            if (TextureImage[loop]->data)                 // If Tex
            {
                free(TextureImage[loop]->data);           // Free T
            }
            free(TextureImage[loop]);                      // Free The Image
        }
    }
    return Status;                                        // Return
}

```

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
    glShadeModel(GL_SMOOTH); // Enable
    glEnable(GL_TEXTURE_2D); // Enable
    return TRUE;
}
```

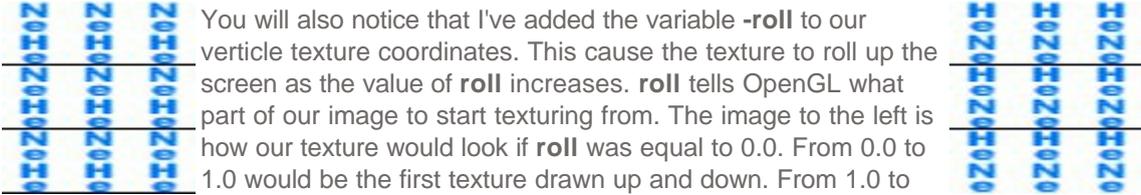
Now for the fun stuff. Our drawing code! We start off the same as usual. We clear the background color and the depth buffer. Then we reset the modelview matrix, and translate into the screen 2 units so that we can see our scene.

```
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.

You will also notice that I've added the variable **-roll** to our vertice 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 texure 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 identical (imagine each marble is a texture). The marble in the center of your infinite number of marbles is your main marble (texture). Its left side is 0.0, its 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.

```
glBindTexture(GL_TEXTURE_2D, texture[0]);           // Select
glBegin(GL_QUADS);                                 // Start !
    glTexCoord2f(0.0f, -roll+0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
    glTexCoord2f(3.0f, -roll+0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
    glTexCoord2f(3.0f, -roll+3.0f); glVertex3f( 1.1f,  1.1f, 0.0f);
    glTexCoord2f(0.0f, -roll+3.0f); glVertex3f(-1.1f,  1.1f, 0.0f);
glEnd();                                           // Done D:
```

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 won't 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.

```
    glBlendFunc(GL_DST_COLOR, GL_ZERO);           // 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.

```
glBindTexture(GL_TEXTURE_2D, texture[3]); // Select
glBegin(GL_QUADS); // Start :
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f,  1.1f, 0.0
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f,  1.1f, 0.0
glEnd(); // Done D
}
```

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.

```

        glBlendFunc(GL_ONE, GL_ONE);
        glBindTexture(GL_TEXTURE_2D, texture[4]);           // Select
        glBegin(GL_QUADS);                                  // Start
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f,  1.1f, 0.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f,  1.1f, 0.0f);
        glEnd();                                           // Done D
    }

```

If **scene** was FALSE, we will draw the first scene (my favorite).

```

    else
    {

```

We start off by checking to see if **masking** is TRUE or 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.

```

            glBindTexture(GL_TEXTURE_2D, texture[1]);       // Select
            glBegin(GL_QUADS);                               // Start
                glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f,
                glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f,
                glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f,  1.1f,
                glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f,  1.1f,
            glEnd();                                         // Done D
        }

```

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.

```

        glBlendFunc(GL_ONE, GL_ONE);
        glBindTexture(GL_TEXTURE_2D, texture[2]);           // Select
        glBegin(GL_QUADS);                                 // Start !
            glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
            glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
            glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f,  1.1f, 0.0f);
            glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f,  1.1f, 0.0f);
        glEnd();                                          // Done D
    }

```

Next we enable depth testing, and disable blending. This prevents strange things from happening in the rest of our program :)

```

    glEnable(GL_DEPTH_TEST);                               // Enable
    glDisable(GL_BLEND);

```

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(
    HINSTANCE Instance,           // Instan
    HINSTANCE PrevInstance,      // Previo
    LPSTR lpCmdLine,            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
    }
}

```

```

// 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.

```

    if (keys[' '] && !sp)
    {
        sp=TRUE; // Tell P:
        scene=!scene;
    }

```

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.

```

if (keys['M'] && !mp)
{
    mp=TRUE;           // Tell P:
    masking=!masking; // Toggle
}

```

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 Fullscreen
    // Recreate Our OpenGL Window
    if (!CreateGLWindow("NeHe's Masking Tutorial",
    {
        return 0; // Quit If Window Was Not Cr
    }
}
}
}
// Shutdown
KillGLWindow();           // Kill The Window
return (msg.wParam);     // Exit The Program
}

```

Creating a mask isn't too 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 FFFFFFFF) 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 a lot 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.
 */

#include <windows.h>                                // Header
#include <stdio.h>                                  // Standard Input ,
#include <stdarg.h>                                  // Header
#include <gl\gl.h>                                   // Header File For
#include <gl\glu.h>                                  // Header
#include <gl\glaux.h>                                // Header

HDC          hDC=NULL;                               // Privat
HGLRC        hRC=NULL;                               // Perman
HWND         hWnd=NULL;
HINSTANCE hInstance;                                // 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          loop1;
int          loop2;
int          delay;
int          adjust=3;                                // Speed .
int          lives=5;                                 // Player
int          level=1;                                  // Intern.
int          level2=level;
int          stage=1;                                  // 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 **x** and **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 **x** and **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      fx, fy;
    int      x, y;
    float    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 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 or 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 :
    unsigned long mm_timer_start;
    unsigned long mm_timer_elapsed;      // Multime
    bool        performance_timer;       // Using '
    __int64     performance_timer_start; // Perform
    __int64     performance_timer_elapsed; // Perform
} 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 };                // Steppi:
```

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); // Declar:
```

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 t

    // 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 Pe
        timer.mm_timer_start = timeGetTime(); // Use ti
        timer.resolution = 1.0f/1000.0f; // Set Ou
        timer.frequency = 1000;
        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 separate 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 performance 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()
{
    __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.

```

    for (loop1=0; loop1<(stage*level); loop1++)           // Loop T
    {
        enemy[loop1].x=5+rand()%6;                       // Select
        enemy[loop1].y=rand()%11;                       // Select
        enemy[loop1].fx=enemy[loop1].x*60;              // Set Fi
        enemy[loop1].fy=enemy[loop1].y*40;              // Set Fi
    }
}

```

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()
{
    int Status=FALSE;                                     // Status
    AUX_RGBImageRec *TextureImage[2];                   // Create
    memset(TextureImage,0,sizeof(void *)*2);           // Set Th

    if          ((TextureImage[0]=LoadBMP("Data/Font.bmp")) &&
                (TextureImage[1]=LoadBMP("Data/Image.bmp"))) // Load B
    {
        Status=TRUE;

        glGenTextures(2, &texture[0]);

        for (loop1=0; loop1<2; loop1++)
        {
            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++)
        {
            if (TextureImage[loop1]) // If Tex
            {
                if (TextureImage[loop1]->data)

```

```

        {
            free(TextureImage[loop1]->data); // Free T
        }
        free(TextureImage[loop1]); // Free T
    }
}
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++) // Loop T
    {
        float cx=float(loop1%16)/16.0f;
        float cy=float(loop1/16)/16.0f;

        glNewList(base+loop1, GL_COMPILE); // Start
            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();
    }
}

```

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 '
{
    char          text[256];
    va_list       ap;

    if (fmt == NULL) // If The
        return;

    va_start(ap, fmt); // Parses
        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
    }

    glCallLists(strlen(text),GL_UNSIGNED_BYTE, text); // Write '
    glDisable(GL_TEXTURE_2D); // Disabl
}

```

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 won't 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);
        glLoadIdentity();                                   // Select
                                                            // 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 anti-aliasing.

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 antialiasing right off the start. Look hard. Notice how the jagged lines on the enemies smooth out when antialiasing is on. The player and hourglass should look better as well.

```

int InitGL(GLvoid)                                           // All Se
{
    if (!LoadGLTextures())
    {

```

```

        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 O
    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)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear
    glBindTexture(GL_TEXTURE_2D, texture[0]); // Select
    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
}

```

If the player still has lives left, we draw animated images of the player's 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 each other. 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 a lot 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++)
{
    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 Pl.
    glBegin(GL_LINES); // Start !
        glVertex2d(-5,-5); // Top Le
        glVertex2d( 5, 5); // Bottom
        glVertex2d( 5,-5); // Top Ri
        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 Pl.
    glBegin(GL_LINES); // Start !
        glVertex2d(-7, 0); // Left C
        glVertex2d( 7, 0); // Right C
        glVertex2d( 0,-7); // Top Ce
        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 too 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;
glLineWidth(2.0f); // Set Li
glDisable(GL_LINE_SMOOTH); // Disabl
glLoadIdentity(); // Reset '
for (loop1=0; loop1<11; loop1++) // Loop F:
{
    for (loop2=0; loop2<11; loop2++) // Loop F:
    {
        glColor3f(0.0f,0.5f,1.0f); // Set Li:
        if (hline[loop1][loop2]) // Has Th
        {
            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 :
                glVertex2d(20+(loop1*60),70+(loop2*40));
                glVertex2d(80+(loop1*60),70+(loop2*40));
            glEnd(); // Done D:
        }
    }
}

```

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 :
            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:



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 =  $\text{loop1}/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f$
- Left X Texture Coordinate =  $\text{loop1}/10 = 0/10 = 0.0f$
- Top Y Texture Coordinate =  $\text{loop2}/10 = 0/10 = 0.0f$ ;
- Bottom Y Texture Coordinate =  $\text{loop2}/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f$ ;

### loop1=1 and loop2=1

- Right X Texture Coordinate =  $\text{loop1}/10+0.1f = 1/10+0.1f = 0.1f+0.1f = 0.2f$
- Left X Texture Coordinate =  $\text{loop1}/10 = 1/10 = 0.1f$
- Top Y Texture Coordinate =  $\text{loop2}/10 = 1/10 = 0.1f$ ;
- Bottom Y Texture Coordinate =  $\text{loop2}/10+0.1f = 1/10+0.1f = 0.1f+0.1f = 0.2f$ ;

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.

```

glEnable(GL_TEXTURE_2D); // Enable
glColor3f(1.0f,1.0f,1.0f); // Bright
glBindTexture(GL_TEXTURE_2D, texture[1]); // Select
if ((loop1<10) && (loop2<10))
{
    // Are All Sides Of The Box Traced?
    if (hline[loop1][loop2] && hline[loop1][loop2+1] && vli
    {
        glBegin(GL_QUADS); // Draw A
            glTexCoord2f(float(loop1/10.0f)+0.1f,
            glVertex2d(20+(loop1*60)+59,(70+loop2
            glTexCoord2f(float(loop1/10.0f),1.0f-
            glVertex2d(20+(loop1*60)+1,(70+loop2*
            glTexCoord2f(float(loop1/10.0f),1.0f-
            glVertex2d(20+(loop1*60)+1,(70+loop2*
            glTexCoord2f(float(loop1/10.0f)+0.1f,
            glVertex2d(20+(loop1*60)+59,(70+loop2
        glEnd(); // Done T
    }
}
glDisable(GL_TEXTURE_2D); // Disabl
}
}
glLineWidth(1.0f); // Set Th

```

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 responsible 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.

```

if (hourglass.fx==1)
{
    glLoadIdentity(); // Reset '
    glTranslatef(20.0f+(hourglass.x*60),70.0f+(hourglass.y*40),0.0f);
    glRotatef(hourglass.spin,0.0f,0.0f,1.0f); // Rotate
    glColor3ub(rand()%255,rand()%255,rand()%255); // Set Ho

```

**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 !
    glVertex2d(-5,-5); // Top Le
    glVertex2d( 5, 5); // Bottom
    glVertex2d( 5,-5); // Top Ri
    glVertex2d(-5, 5); // Bottom
    glVertex2d(-5, 5); // Bottom
    glVertex2d( 5, 5); // Bottom
    glVertex2d(-5,-5); // Top Le
    glVertex2d( 5,-5); // Top Ri
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.

```

glLoadIdentity(); // Reset '
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 Pl
glBegin(GL_LINES); // Start :
    glVertex2d(-5,-5); // Top Le
    glVertex2d( 5, 5); // Bottom
    glVertex2d( 5,-5); // Top Ri
    glVertex2d(-5, 5); // Bottom
glEnd(); // Done D

```

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); // Set Pl
glBegin(GL_LINES); // Start :
    glVertex2d(-7, 0); // Left C
    glVertex2d( 7, 0); // Right C
    glVertex2d( 0,-7); // Top Ce
    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++) // Loop Th
{
    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 :
        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
    glEnd(); // Done D:
}

```

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,0.0f); // Make E:
        glBegin(GL_LINES); // Start :
            glVertex2d(-7,-7); // Top Le
            glVertex2d( 7, 7); // Bottom
            glVertex2d(-7, 7); // Bottom
            glVertex2d( 7,-7); // Top Ri
        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 :
    {
        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(
    HINSTANCEhInstance, // Instan
    HINSTANCEhPrevInstance, // Previo
    LPSTR lpCmdLine, // Window
    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
    }
}

```

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:
        {

```

```

        if (msg.message==WM_QUIT)                // Have W
        {
            done=TRUE;
        }
        else
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
else
{

```

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.

```

float start=TimerGetTime();                    // Grab T

// 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

while(TimerGetTime()<start+float(steps[adjust]*2.0f)) {} // Waste

```

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 I
{
    keys[VK_F1]=FALSE;                         // If So I
    KillGLWindow();
}

```

```

        fullscreen=!fullscreen;
        // Recreate Our OpenGL Window
        if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,f
        {
            return 0; // Quit I
        }
    }
}

```

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.

```

    if (keys['A'] && !ap)
    {
        ap=TRUE; // ap Bec
        anti=!anti;
    }
    if (!keys['A'])
    {
        ap=FALSE; // ap Bec
    }
}

```

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 not 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 won't 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**.

```

    if (!gameover && active) // If Gam
    {
        for (loop1=0; loop1<(stage*level); loop1++) // Loop T
        {

```

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 player's **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 **x** position is less than the player **x** position, and the enemy's fine **y** position is lined up with a horizontal line, we move the enemy **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 **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 **x** and **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 **x** and **y** positions just tells our program where we WANT the enemies to move.

```

if ((enemy[loop1].x<player.x) && (enemy[loop1]
{
    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]
{
    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**).

```

if (delay>(3-level) && (hourglass.fx!=2))
{
    delay=0;
    for (loop2=0; loop2<(stage*level); lc
    {

```

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.

```

        if (enemy[loop2].fx < enemy[x] * 60)
        {
            enemy[loop2].fx += steps[adjust];
            enemy[loop2].spin += steps[adjust];
        }
        if (enemy[loop2].fx > enemy[x] * 60)
        {
            enemy[loop2].fx -= steps[adjust];
            enemy[loop2].spin -= steps[adjust];
        }
        if (enemy[loop2].fy < enemy[y] * 40)
        {
            enemy[loop2].fy += steps[adjust];
            enemy[loop2].spin += steps[adjust];
        }
        if (enemy[loop2].fy > enemy[y] * 40)
        {
            enemy[loop2].fy -= steps[adjust];
            enemy[loop2].spin -= steps[adjust];
        }
    }
}

```

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.

```

// Are Any Of The Enemies On Top Of The Player
if ((enemy[loop1].fx==player.fx) && (enemy[loc
{
    lives--; // If So,

    if (lives==0)
    {
        gameover=TRUE;
    }

    ResetObjects();
    PlaySound("Data/Die.wav", NULL, SND_S
}
}
}

```

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 too 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 won't be going off the left side of the grid. When moving down we make sure the player won't 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.

```

if (keys[VK_RIGHT] && (player.x<10) && (player.fx==play
{
    hline[player.x][player.y]=TRUE;
    player.x++;
}
if (keys[VK_LEFT] && (player.x>0) && (player.fx==player
{
    player.x--;
    hline[player.x][player.y]=TRUE;
}
if (keys[VK_DOWN] && (player.y<10) && (player.fx==playe
{
    vline[player.x][player.y]=TRUE;
    player.y++;
}

```

```

    if (keys[VK_UP] && (player.y>0) && (player.fx==player.x
    {
        player.y--;
        vline[player.x][player.y]=TRUE;
    }

```

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) // Is Fin
    {
        player.fx+=steps[adjust]; // If So,
    }
    if (player.fx>player.x*60) // Is Fin
    {
        player.fx-=steps[adjust]; // If So,
    }
    if (player.fy<player.y*40) // Is Fin
    {
        player.fy+=steps[adjust]; // If So,
    }
    if (player.fy>player.y*40) // Is Fin
    {
        player.fy-=steps[adjust]; // 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;           // Displa
        stage=0;           // Game S
        lives=5;          // 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 level (**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);
        stage++;           // Increa
        if (stage>3)
        {
            stage=1;      // If So,
            level++;      // Increa
            level2++;     // Increa
            if (level>3)
            {
                level=3;  // If So,
                lives++;  // Give T
                if (lives>5)
                {
                    lives=5; // If So,
                }
            }
        }
    }
}

```

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.

```

ResetObjects();

```

```

        for (loop1=0; loop1<11; loop1++)          // Loop T
        {
            for (loop2=0; loop2<11; loop2++)      // Loop T
            {
                if (loop1<10)
                {
                    hline[loop1][loop2]=FALSE;
                }
                if (loop2<10)
                {
                    vline[loop1][loop2]=FALSE;
                }
            }
        }
    }
}

```

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.

```

// If The Player Hits The Hourglass While It's Being Displayed O
if ((player.fx==hourglass.x*60) && (player.fy==hourglass.y*40) &
{
    // Play Freeze Enemy Sound
    PlaySound("Data/freeze.wav", NULL, SND_ASYNC | SND_LOOP
    hourglass.fx=2;
    hourglass.fy=0;
}
}

```

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.

```

player.spin+=0.5f*steps[adjust];          // Spin T
if (player.spin>360.0f)
{
    player.spin-=360;                      // If So,
}

```

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.

```
hourglass.spin-=0.25f*steps[adjust];           // Spin T
if (hourglass.spin<0.0f)                       // Is The
{
    hourglass.spin+=360.0f;
}
```

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 won't 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.

```
hourglass.fy+=steps[adjust];
if ((hourglass.fx==0) && (hourglass.fy>6000/level)) // Is The
{
    PlaySound("Data/hourglass.wav", NULL, SND_ASYNC);
    hourglass.x=rand()%10+1;                       // Give T
    hourglass.y=rand()%11;
    hourglass.fx=1;
    hourglass.fy=0;
}
```

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.

```
if ((hourglass.fx==1) && (hourglass.fy>6000/level)) // Is The
{
    hourglass.fx=0;
    hourglass.fy=0;
}
```

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.

```

if ((hourglass.fx==2) && (hourglass.fy>500+(500*level)))// Is Th
{
    PlaySound(NULL, NULL, 0);           // If So,
    hourglass.fx=0;
    hourglass.fy=0;
}

```

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++;                       // Increa
    }
}

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

```

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>
#include <stdio.h> // Header
#include <gl\gl.h> // Header
#include <gl\glu.h>
#include <gl\glaux.h>
#include "glext.h" // Header
#include <string.h>
#include <math.h> // Header
```

The **GLfloat MAX\_EMOSS** 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_EMOSS (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.

```
#define __ARB_ENABLE true // Used To
// #define EXT_INFO
#define MAX_EXTENSION_SPACE 10240 // Character
#define MAX_EXTENSION_LENGTH 256 // Maximum
bool multitextureSupported=false; // Flag I:
bool useMultitexture=true; // 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 **glMultiTexCoordfARB** map to the well-known **glTexCoordf**, specifying i-dimensional texture-coordinates. Note that these can totally substitute the **glTexCoordf**-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 multitexture-extension 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 omitted 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.

```

GLuint filter=1; // Which I
GLuint texture[3];
GLuint bump[3]; // Our Bu
GLuint invbump[3];
GLuint glLogo;
GLuint multiLogo; // Handle
GLfloat LightAmbient[] = { 0.2f, 0.2f, 0.2f};
GLfloat LightDiffuse[] = { 1.0f, 1.0f, 1.0f};
GLfloat LightPosition[] = { 0.0f, 0.0f, 2.0f};
GLfloat Gray[] = { 0.5f, 0.5f, 0.5f, 1.0f};

```

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 Tessellation 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); // Declared

```

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’-separated 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++) {
        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 **wglGetProcAddress()**-calls with a string naming the function call as parameter and a prototype-cast to ensure we’ll get the correct function type.

```

bool initMultitexture(void) {
    char *extensions;
    extensions=(char *) glGetString(GL_EXTENSIONS);
}

```

```

    int len=strlen(extensions);
    for (int i=0; i<len; i++) // Separat
        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) wglGetProcAddress

#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; // Status
    AUX_RGBImageRec *Image=NULL;
    char *alpha=NULL;
}

```

```

// 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
    glPixelTransferf(GL_GREEN_SCALE, 0.5f);
    glPixelTransferf(GL_BLUE_SCALE, 0.5f);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP); // No Wrap
    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];

        glGenTextures(3, invbump); // Create

        // 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];
        alpha[4*a+1]=Image->data[a*3+1]; // G
        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 (Image->data) delete Image->data; // If Tex
        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 (Image->data) delete Image->data; // If Tex
        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

```

```

        glVertex3f( 0.0f,-1.0f, 0.0f);
        for (i=12; i<16; i++) {
            glTexCoord2f(data[5*i],data[5*i+1]);
            glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
        }
        // Right Face
        glVertex3f( 1.0f, 0.0f, 0.0f);
        for (i=16; i<20; i++) {
            glTexCoord2f(data[5*i],data[5*i+1]);
            glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
        }
        // Left Face
        glVertex3f(-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
{
    multitextureSupported=initMultitexture();
    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

    initLights();
    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: <http://www.objectecture.com/> 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

$$C=(L*N) \times D_l \times D_m$$

- **L** is light vector
- **N** is normal vector
- **D<sub>l</sub>** is light diffuse color
- **D<sub>m</sub>** 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 **F<sub>d</sub>**
- **(F<sub>d</sub>+m)** approximates **(L\*N)** per pixel

### Approximate Derivative

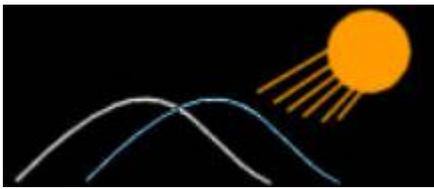
#### Embossing Approximates Derivative

- Lookup height **H<sub>0</sub>** at point **(s,t)**
- Lookup height **H<sub>1</sub>** at point slightly perturbed toward light source **(s+ds,t+dt)**
- Subtract original height **H<sub>0</sub>** from perturbed height **H<sub>1</sub>**
- Difference represents instantaneous slope **m=H<sub>1</sub>-H<sub>0</sub>**

### Compute The Bump



1) Original bump (**H<sub>0</sub>**).



2) Original bump ( $H_0$ ) overlaid with second bump ( $H_1$ ) slightly perturbed toward light source.



3) Subtract original bump from second ( $H_0 - H_1$ ). This leads to brightened (**B**) and darkened (**D**) areas.

### Compute The Lighting

#### Evaluate Fragment Color $C_f$

- $C_f = (L \cdot N) \times D_l \times D_m$
- $(L \cdot N) \sim (F_d + (H_1 - H_0))$
- $D_m \times D_l$  is encoded in surface texture  $C_t$ . Could control  $D_l$  separately, if you're clever. (we control it using OpenGL-Lighting!, **rem.: Jens**)
- $C_f = (F_d + (H_0 - H_1)) \times C_t$

### 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  $F_d$  (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  $M_n$  from axes
- Transform light vector into normal space. ( $M_n$  is also called an orthonormal basis. Think of  $M_n \cdot v$  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
- **H0** 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 built-in 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.

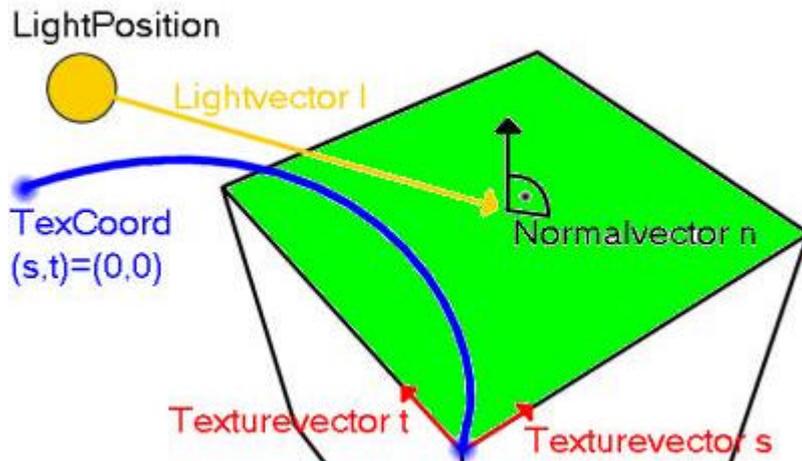
- We calculate the **current vertex c** on our surface (simply by looking it up in **data**).
- Then we'll calculate a normal **n** with length 1 (We usually know **n** for each face of a cube!). This is important, since we can save computing time by requesting normalized vectors. Calculate the **light 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\_EBBOSS**: **ds = s\*v\*MAX\_EBBOSS**, **dt=t\*v\*MAX\_EBBOSS**. Note that **s**, **t** and **v** are vectors while **MAX\_EBBOSS** 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]=l[0]-c[0];
v[1]=l[1]-c[1];
v[2]=l[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[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_EBBOSS;
c[1]=(t[0]*v[0]+t[1]*v[1]+t[2]*v[2])*MAX_EBBOSS;

```

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 'c'
    GLfloat n[4]={0.0f,0.0f,0.0f,1.0f}; // Normal
    GLfloat s[4]={0.0f,0.0f,0.0f,1.0f}; // s-Texture
    GLfloat t[4]={0.0f,0.0f,0.0f,1.0f}; // t-Texture
    GLfloat l[4];
    GLfloat Minv[16]; // Holds 'Minv'
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear '

    // Build Inverse Modelview Matrix First. This Substitutes One Push/Pop With One gl
    // 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];
    l[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++) {
    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 texel-units. 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!

```

bool doMesh2TexelUnits(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 l[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 gl
// 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];
l[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 **glMultiTexCoord2fARB()** 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++) {
    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 texture-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:

```

bool DrawGLScene(GLvoid) // Here's
{
    if (bumps) {
        if (useMultitexture && maxTexelUnits>1)
            return doMesh2TexelUnits();
        else return doMesh1TexelUnits();
    }
    else return doMeshNoBumps();
}

```

Kills the GLWindow, not modified (thus omitted):

```

GLvoid KillGLWindow(GLvoid) // Proper
>...<

```

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):

```

LRESULT CALLBACK WndProc(   HWND hWnd,                               // Handle
                           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,           // Previous Instance
                           LPSTR lpCmdLine,                   // Command Line
                           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])

```

```

        {
            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 let's 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; // Our Mo

POINT mpos; // Mouse :

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); // Swap Buffers (D
GetCursorPos(&mpos); // Get Th
SetCursorPos(320,240); // Set Mo
heading += (float)(320 - mpos.x)/100 * 5; // Update The Direc
yrot = heading; // Update
lookupdown -= (float)(240 - mpos.y)/100 * 5; // Update The X Rot
```

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.

1. 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.
2. 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> // Header File For
#include <gl\gl.h> // Header File For
#include <gl\glu.h> // Header
#include <gl\glaux.h> // Header
#include <dinput.h> // 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
    if ( DirectInputCreateEx( hInstance, // Window Instance
                             DIRECTINPUT_VERSION, // Direct
                             IID_IDirectInput7, // Version 7
                             (void*)&g_DI, // Direct
                             NULL ) // NULL Parameter
    {
        return(false); // Couldn
    }

    // Create The Keyboard Device
    if ( g_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 ( g_KDIDev->SetDataFormat(&c_dfDIKeyboard) )
    {
        return(false); // Couldn
    }

    // Set The Cooperative Level
    if ( g_KDIDev->SetCooperativeLevel(hWnd, DISCL_FOREGROUND | DISCL_EXCLUSIVE) )
    {
```

```

        return(false); // Could I
    }

    if (g_KDIDev) // Did We
        g_KDIDev->Acquire(); // If So,
    else // If Not
        return(false); // Return

    return(true); // Everythi
}

// 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

```

        case WM_KEYDOWN: // Is A Key Being I
        {
            keys[wParam] = TRUE; // If So,
            return 0; // Jump Back
        }

        case WM_KEYUP: // Has A I
        {
            keys[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
bool    fp; // F1 Key

...

GLfloat lookupdown = 0.0f;

```

```

GLfloat  z=0.0f;                                     // Depth :
GLuint   filter;                                    // Which :

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 I
}

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)
    {
        bp = true;                                  // Is The
        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 Up
{
    z-=0.02f;
}

if ( buffer[DIK_NEXT] & 0x80 )                     // Page D
{
    z+=0.02f;
}

```

```

if ( buffer[DIK_UP] & 0x80 ) // Up Arrow
{
    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 * piover180) * 0.05f;
}

if ( buffer[DIK_DOWN] & 0x80 ) // Down Arrow
{
    xpos += (float)sin(heading*piover180) * 0.05f;
    zpos += (float)cos(heading*piover180) * 0.05f;
    if (walkbiasangle <= 1.0f)
    {
        walkbiasangle = 359.0f;
    }
    else
    {
        walkbiasangle-= 10;
    }

    walkbias = (float)sin(walkbiasangle * piover180) * 0.05f;
}

if ( buffer[DIK_LEFT] & 0x80 ) // Left Arrow
{
    xpos += (float)sin((heading - 90)*piover180) * 0.05f;
    zpos += (float)cos((heading - 90)*piover180) * 0.05f;
    if (walkbiasangle <= 1.0f)
    {
        walkbiasangle = 359.0f;
    }
    else
    {
        walkbiasangle-= 10;
    }

    walkbias = (float)sin(walkbiasangle * piover180) * 0.05f;
}

if ( buffer[DIK_RIGHT] & 0x80 ) // Right Arrow
{
    xpos += (float)sin((heading + 90)*piover180) * 0.05f;
    zpos += (float)cos((heading + 90)*piover180) * 0.05f;
    if (walkbiasangle <= 1.0f)
    {
        walkbiasangle = 359.0f;
    }
    else
    {
        walkbiasangle-= 10;
    }

    walkbias = (float)sin(walkbiasangle * piover180) * 0.05f;
}

if ( buffer[DIK_F1] & 0x80) // Is F1 Being Pressed

```

```

        {
            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 I
                    }

                if (!DI_Init()) // ReInit
                {
                    return 0;// Couldn't Initia
                }
            }
        }
    else
    {
        fp = false; // Set 'f
    }
}

// Shutdown
DX_End(); // Destroys Direct
KillGLWindow(); // Kill T
return (msg.wParam); // Exit T
}

```

#### 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++)
{
    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; // Mouse :
int      adjust = 5; // Speed :

// Create A Structure For The Timer Information ( Add )

```

```

struct
{
    __int64      frequency;                // Timer :
    float        resolution;              // Timer :
    unsigned long mm_timer_start;         // Multime
    unsigned long mm_timer_elapsed;       // Multimedia Time:
    bool         performance_timer;       // Using The Perfo
    __int64      performance_timer_start; // Performance Time
    __int64      performance_timer_elapsed; // Performance Time
} timer;                                  // Structure Is Nar

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 Th
    }
    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
    }
    else // Not Ti
    {
        while(TimerGetTime()<start+float(adjust*2.0f)) {}
    }

```

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

```

typedef struct tagTRIANGLE
{
    int texture; ( Add )
    VERTEX vertex[3];
} TRIANGLE;

```

#### 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++)
    {

```

#### Modify the DrawGLScene code

```

// Process Each Triangle
for (int loop_m = 0; loop_m < numtriangles; loop_m++)

```

```

    {
        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
{
    int Status=FALSE; // Status Indicator
    AUX_RGBImageRec *TextureImage[5]; // Create Storage
    memset(TextureImage,0,sizeof(void *)*2); // Set The Pointer
    if ( (TextureImage[0]=LoadBMP("Data/floor1.bmp")) && // Load T
        (TextureImage[1]=LoadBMP("Data/light1.bmp")) && // Load T
        (TextureImage[2]=LoadBMP("Data/rustyblue.bmp")) && // Load the Wall Te
        (TextureImage[3]=LoadBMP("Data/crate.bmp")) && // Load T
        (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++) // Loop Through 5
        {
            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++) // Loop Th
        {
            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 a lot 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]; //
```

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() //
{
    int Status=FALSE; // Status :
    AUX_RGBImageRec *TextureImage[2]; // Create :
    memset(TextureImage,0,sizeof(void *)*2); // 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")) && //
    (TextureImage[1]=LoadBMP("Data/Reflect.bmp"))) //
{
    Status=TRUE; //

    glGenTextures(6, &texture[0]); //

    for (int loop=0; loop<=1; loop++)
    {
        // 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, Tex

        // Create Linear Filtered Texture
        glBindTexture(GL_TEXTURE_2D, texture[loop+2]); //
        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_MIP
        gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[loop]->sizeX, 1
    }
    for (loop=0; loop<=1; loop++)
    {
        if (TextureImage[loop]) //
        {
            if (TextureImage[loop]->data) //
            {
                free(TextureImage[loop]->data); //
            }
            free(TextureImage[loop]); // Free The
        }
    }
}

return Status; //
}
```

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) )

```
        glVertex3f( 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); // Set The
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // 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) //
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Th
    glLoadIdentity(); // Reset Th

    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);
    gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // A Cylinder
    break;
case 2:
    gluSphere(quadratic,1.3f,32,32); // Sphere
    break;
case 3:
    glTranslatef(0.0f,0.0f,-1.5f);
    gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32); // Cone With
    break;
};

glPopMatrix();
glDisable(GL_TEXTURE_GEN_S); //
glDisable(GL_TEXTURE_GEN_T); //

glBindTexture(GL_TEXTURE_2D, texture[filter*2]); // This Will
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, 10.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 13.3f, 10.0f, 10.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-13.3f, 10.0f, 10.0f);
glEnd();

glPopMatrix();

xrot+=xspeed;
yrot+=yspeed;
return TRUE; //
}
```

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 couldn't 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)**

[tiptup@net4tv.com](mailto: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.

```
#include <windows.h>    // Header File For Windows
#include <stdio.h>      // Header File For Standard Input / Output
#include <stdarg.h>     // Header File For Variable Argument Routines
#include <string.h>     // Header File For String Management
#include <gl\gl.h>      // Header File For The OpenGL32 Library
#include <gl\glu.h>     // Header File For The GLu32 Library

HDC      hDC=NULL;    // Private GDI Device Context
HGLRC    hRC=NULL;    // Permanent Rendering Context
HWND     hWnd=NULL;   // Holds Our Window Handle
HINSTANCE hInstance; // Holds The Instance Of The Application

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
```

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 genius. I'd like to point out that I did not write all of this code on my own. A lot 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(TextureImage *texture, char *filename) // Loads A TGA File Into Memory
{
    GLubyte TGAheader[12]={0,0,2,0,0,0,0,0,0,0,0,0}; // Uncompressed TGA Header
    GLubyte TGAcompare[12]; // Used To Compare TGA Header
```

## Jeff Molofee's OpenGL Windows Tutorial #25

```
GLubyte    header[6];    // First 6 Useful Bytes From The Header
GLuint     bytesPerPixel; // Holds Number Of Bytes Per Pixel Used In The TGA File
GLuint     imageSize;   // Used To Store The Image Size When Setting Aside Ram
GLuint     temp;        // Temporary Variable
GLuint     type=GL_RGBA; // Set The Default GL Mode To RGBA (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
```

## Jeff Molofee's OpenGL Windows Tutorial #25

```
(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 Ma
{
    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.

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 In 'temp' (1st B)
    texture->imageData[i + 2] = temp; // Set The 3rd Byte To The Value In 'temp' (1st B)
}

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 won't 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

## Jeff Molofee's OpenGL Windows Tutorial #25

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++)    // 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.

## Jeff Molofee's OpenGL Windows Tutorial #25

```
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 Happens
{
    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 Window
{
    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.

## Jeff Molofee's OpenGL Windows Tutorial #25

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 1,64 (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).

## Jeff Molofee's OpenGL Windows Tutorial #25

```
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( 0,128); // Bottom Left Of Top Box
    glVertex2d(639,128); // Bottom Right Of Top Box
    glVertex2d(639, 1); // Top Right Of Top Box
    glVertex2d( 0, 1); // Top Left Of Top Box
    glVertex2d( 0,417); // Down To Top Left Of Bottom Box
glEnd(); // Done Second Line Strip
```

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 It
```

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, Separated 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 Extensio
```

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?
    {
        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_ICONINFORMATION);
        }

        if (!wglDeleteContext(hRC))    // Are We Able To Delete The RC?
        {
            MessageBox(NULL,"Release Rendering Context Failed.", "SHUTDOWN ERROR", MB_OK | MB_ICONINFORMATION);
        }
        hRC=NULL;    // Set RC To NULL
    }
}
```

## Jeff Molofee's OpenGL Windows Tutorial #25

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

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
                  HINSTANCE hPrevInstance, // Previous Instance
                  LPSTR lpCmdLine, // Command Line Parameters
                  int nCmdShow) // Window Show State
{
    MSG msg; // Windows Message Structure
    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_YESNO) <= ID_NO)
    {
        fullscreen=FALSE; // Windowed Mode
    }

    // Create Our OpenGL Window
    if (!CreateGLWindow("NeHe's Token, Extensions, Scissoring & TGA Loading Tutorial",640,480))
    {
        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?
            {
                done=TRUE; // If So done=TRUE
            }
            else // If Not, Deal With Window Messages
            {
                DispatchMessage(&msg); // Dispatch The Message
            }
        }
        else // If There Are No Messages
        {
            // Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
            if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit Message?
            {
                done=TRUE; // ESC or DrawGLScene Signalled A Quit
            }
            else // Not Time To Quit, Update Screen
            {
                UpdateScreen();
            }
        }
    }
}
```

## Jeff Molofee's OpenGL Windows Tutorial #25

```
{
    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 Tut
        {
            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  $(32 * (\text{maxtokens} - 9))$  **scroll** will be increased by 2, and 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.

```
if (keys[VK_DOWN] && (scroll<32*(maxtokens-9)))    // Is Down Arrow Being Pressed?
{
    scroll+=2;    // If So, Increase 'scroll' Moving Screen Up
}
}
}
}

// Shutdown
KillGLWindow();    // Kill The Window
return (msg.wParam);    // Exit The Program
}
```

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!

```
#include <windows.h>          // Header File For Windows
#include <math.h> // Math Library Header File
#include <stdio.h>           // Header File For Standard Input/Output
#include <gl\gl.h>           // Header File For The OpenGL32 Library
#include <gl\glu.h>          // Header 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.

## Jeff Molofee's OpenGL Windows Tutorial #26

```
GLfloat      xrot,yrot,zrot,    // X, Y & Z Rotation
            xspeed,yspeed,zspeed, // X, Y & Z Spin Speed
            cx,cy,cz=-15;      // X, Y & Z Position

int          key=1;           // Used To Make Sure Same Morph Key Is Not Pressed
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      x, y, z; // X, Y & Z Points
} VERTEX;      // 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 vertices (**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       morph1,morph2,morph3,morph4, // Our 4 Morphable Objects (morph1,2,
            helper,*sour,*dest;           // Helper Object, Source Obj
```

Same as always, we declare WndProc().

## Jeff Molofee's OpenGL Windows Tutorial #26

```
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 object (**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 by 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 // Do This
    {
        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 Reached
    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
```

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 Vertices (x, y and z values)!

After reading in all of the vertices 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 vertices, **maxver** will become 20. If we read in another object, and it has 40 vertices, **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 **a.x**=(40-20)/200... **a.x**=(20)/200... **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.

```

VERTEX calculate(int i)    // Calculates Movement Of Points During Morphing
{
    VERTEX a;              // Temporary Vertex Called a
    a.x=(sour->points[i].x-dest->points[i].x)/steps; // a.x Value Equals Source x
    a.y=(sour->points[i].y-dest->points[i].y)/steps; // a.y Value Equals Source y
    a.z=(sour->points[i].z-dest->points[i].z)/steps; // a.z Value Equals Source z
    return a;              // Return The Results
} // This Makes Points Move At A Speed So They All Get To Their

```

The ReSizeGLScene() code hasn't changed so we'll skip over it.

```

GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Windo

```

## Jeff Molofee's OpenGL Windows Tutorial #26

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, 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 we'll 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 File
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(&morph4,468); // Manually Reserver Ram For A 4th 468 Vertice Object (
for(int i=0;i<468;i++) // Loop Through All 468 Vertices
{
    morph4.points[i].x=((float)(rand()%14000)/1000)-7; // morph4 x Point B
    morph4.points[i].y=((float)(rand()%14000)/1000)-7; // morph4 y Point B
    morph4.points[i].z=((float)(rand()%14000)/1000)-7; // morph4 z Point B
}
```

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
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
    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 xspeed, yspeed & zspeed

    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**. (**tx/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 morph1
{
    // The Same Amount Of Verts For Simplicity, Could Use maxver Also
    if(morph) q=calculate(i); else q.x=q.y=q.z=0; // If morph is true calculate the movement
    helper.points[i].x-=q.x; // Subtract q.x Units From helper.points[i].x
    helper.points[i].y-=q.y; // Subtract q.y Units From helper.points[i].y
    helper.points[i].z-=q.z; // Subtract q.z Units From helper.points[i].z
    tx=helper.points[i].x; // Make Temp X Variable Equal To helper.points[i].x
    ty=helper.points[i].y; // Make Temp Y Variable Equal To helper.points[i].y
    tz=helper.points[i].z; // Make Temp Z Variable Equal To helper.points[i].z
```

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 C
    // 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;}
}

```

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",
        }
    }
}

```

## Jeff Molofee's OpenGL Windows Tutorial #26

```
        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 |
        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_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 :)

```
int WINAPI WinMain(           HINSTANCE     hInstance,           // Instance
                           HINSTANCE     hPrevInstance,       // Previous Instance
                           LPSTR         lpCmdLine,           // Command Line Parameters
                           int           nCmdShow)           // Window Show State
{
    MSG     msg;           // Windows Message Structure
    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 FullScreer
    {
        fullscreen=FALSE; // Windowed Mode
    }

    // Create Our OpenGL Window
    if (!CreateGLWindow("Piotr Cieslak & NeHe's Morphing Points Tutorial",640,480,16,i
    {
        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?
            {
                done=TRUE;           // If So done=TRUE
            }
        }
    }
}
```



```

if (keys['Z']) // Is Z Key Being Pressed?
    cz+=0.01f; // Move Object Towards Viewer

if (keys['W']) // Is W Key Being Pressed?
    cy+=0.01f; // Move Object Up

if (keys['S']) // Is S Key Being Pressed?
    cy-=0.01f; // Move Object Down

if (keys['D']) // Is D Key Being Pressed?
    cx+=0.01f; // Move Object Right

if (keys['A']) // Is A Key Being Pressed?
    cx-=0.01f; // 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
    dest=&morph1; // Destination Object To Mor
}
if (keys['2'] && (key!=2) && !morph) // Is 2 Pre
{
    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
}
if (keys['4'] && (key!=4) && !morph) // Is 4 Pre
{
    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!

```

if (keys[VK_F1]) // Is F1 Being Pressed?
{
    keys[VK_F1]=FALSE; // If So Make Key F1
    KillGLWindow(); // Kill Our Current Window
    fullscreen=!fullscreen; // Toggle Fullscreen
// Recreate Our OpenGL Window
    if (!CreateGLWindow("Piotr Cieslak & NeHe's Mo

```

```
        {
            return 0;          // Quit If Window W:
        }
    }
}

// Shutdown
KillGLWindow(); // Kill The Window
return (msg.wParam); // Exit The Program
}
```

I hope you have enjoyed this tutorial. Although it's not an incredibly complex tutorial, you can learn a lot 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. A lot 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](#) )

# 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 myself 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 fi

## What You'll Need To Get Started

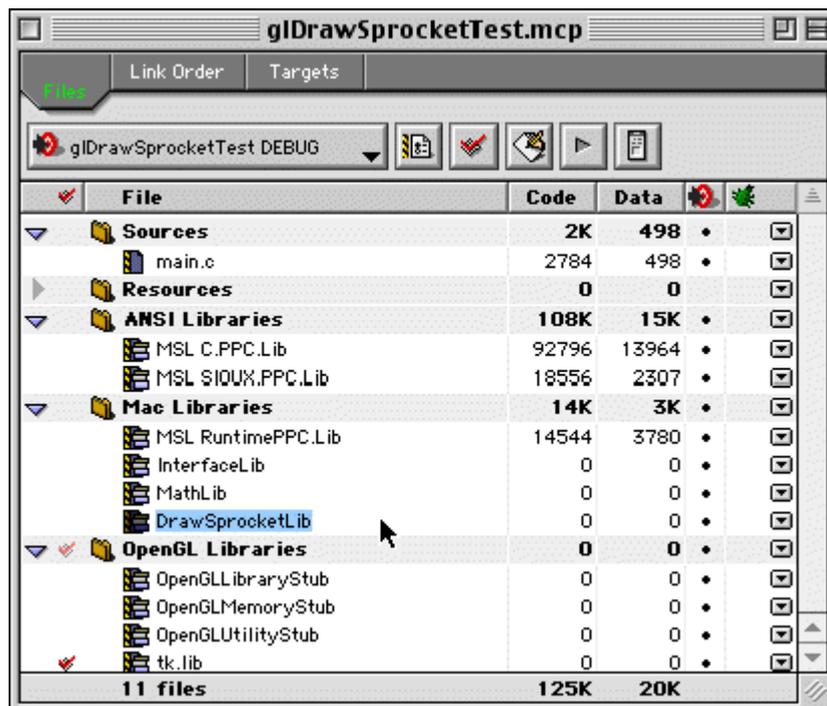
I know, you're getting tired of listening to me, so on to the tutorial. Basically you should already know how to do MacOS out [Macintosh C](#). You will also need [Apple's OpenGL SDK 1.0](#) and the [DrawSprocket SDK](#). If you don't already know h which has tons of OpenGL tutorials (including MacOS ports). Oh, and don't forget a compiler, if you don't have one I s the Discovery Programming edition) or if you don't want to spend any money, or don't have any to spend, you can get [I](#) 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):

- OpenGGLibraryStub
- 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:



(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>                // ANSI C cross platform headers
#include <stdio.h>
#include <Types.h>                // Standard MacOS Headers
#include <Memory.h>
#include <Quickdraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Menus.h>
#include <Windows.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <OSUtils.h>
#include <ToolUtils.h>
#include <SegLoad.h>
#include <Sound.h>
#include <DrawSprocket.h>        // DrawSprocket
#include <agl.h>                // Apple's OpenGL
#include <glu.h>                // Used for setting perspective and making objects
#include <tk.h>                 // Used for loading images

```

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              640
#define SCREEN_HEIGHT            480

// Texture filters
#define NEAREST                   0
#define LINEAR                   1
#define MIPMAP                   2

```

Most of these won't make sense until you see them when they are used, but I'll describe them now anyway. kMoveToFront 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 :

```

/**> GLOBAL VARIABLES <*/
DSpContextAttributes             gDSpContextAttributes;    // Global DrawSprocket context attril
DSpContextReference              gDSpContext;             // The global DrawSprocket context
AGLContext                       gOpenGLContext;         // The global OpenGL (AGL) context
// Texture Maps
GLuint                           gTutorialTexture;
// Lighting Info
GLfloat                          gLightAmbient[] = { 0.5, 0.5, 0.5, 1.0 };
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 included with gDspContextAttributes, it'll get used to return the screen to its original resolution and bit depth when the application quits. Things such as color depth, depth buffer depth, and more, this will get explained more later. gLightAmbient, gLightDiffuse, and gLightPosition work because there are better tutorials out there which can teach you this). gMaterialAmbient, gMaterialDiffuse, and gMaterialSpecular are also used for the lighting, but describes the objects 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. You'll have to write a tutorial, but there are those out there which might want the extra explanation. 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 );
void            CleanupAGL( AGLContext context );
void            Reshape3D( int w, int h );
void            InitGL( 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 on the screen, creates a new DrawSprocket context, creates a new window, and then fades back in. CreateWindow() is called for the application window and returns the screen to its original state. SetupAGL() creates a new AGL context which is used by OpenGL and the like. CleanupAGL() gets rid of an AGL context. InitGL() initializes OpenGL things such as depth testing, backface culling, and the like. DrawGL() does all the OpenGL drawing and swaps the offscreen buffer at the end. LoadGLTexture() is a function which I

### The main() Function

```

/*****> main() <*****/
void            main( void )
{
    CGrafPtr theScreen;

    // Do a bunch of MacOS Inits
    ToolboxInit();

```

The call to ToolboxInit() will prepare our application to work as a MacOS application, if we don't call it our application will not be able to 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() returns the screen to the application.

```

// Setup the OpenGL context
gOpenGLContext = SetupAGL( ( AGLDrawable )theScreen );

```

```

if ( !gOpenGLContext )
    return;
Reshape3D( SCREEN_WIDTH, SCREEN_HEIGHT );

```

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.

```

// Event Loop
while ( !Button() )
    DrawGL( gOpenGLContext );

```

Here we will keep drawing our OpenGL scene until the mouse button is pressed. This is where you would normally put ; 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

```

/*****> ToolboxInit() <*****/
void ToolboxInit( void )
{
    MaxApplZone();

    InitGraf( &qd.thePort );
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs( 0L );
    InitCursor();
}

```

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

```

/*****> SetupScreen() <*****/
CGrafPtr SetupScreen( void )
{
    OSStatus theError;
    CGrafPtr theFrontBuffer;

    // Start DrawSprocket
    theError = DSpStartup();
    if ( theError )
        DebugStr( "\pUnable to startup\n" );

```

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 necessarily 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'll have problems and won't be able to create a context at all.

```

// Find the best context for our attributes
theError = DSpFindBestContext( &gDspContextAttributes, &gDspContext );
if ( theError )
    DebugStr("\pUnable to find a suitable device\n");

```

DSpFindBestContext() tries to find a context that closely matches the attributes that we pass to it.

```

// Reserve that context
theError = DSpContext_Reserve( gDspContext, &gDspContextAttributes );
if ( theError )
    DebugStr("\pUnable to create the display!");

```

DSpContext\_Reserve() reserves the context for our application.

```

// Fade out
theError = DSpContext_FadeGammaOut( NULL, NULL );
if( theError )
    DebugStr("\pUnable to fade the display!");

```

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

```

theError = DSpContext_SetState( gDspContext, kDspContextState_Active );
if ( theError )
    DebugStr("\pUnable to set the display!");

```

Now we've set the screen to the state of the new context we created.

```
// Fade in
theError = DSpContext_FadeGammaIn( NULL, NULL );
if ( theError )
    DebugStr("\pUnable to fade the display!");
```

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 separate

```
    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, kMoveToFront
```

NewCWindow() creates a new color window which we store in our color graphics port (theFrontBuffer), we pass in the 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 window is created
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 window is shown, the screen is black (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 need to show the window and set 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 foreground drawing color to that listed at the top of the function (black) and the background color to that listed at the bottom (white).

### The ShutdownScreen() Function

```

/*****> ShutdownScreen() <*****/
void ShutdownScreen( CGrafPtr theFrontBuffer )
{
    DSpContext_FadeGammaOut( NULL, NULL );
    DisposeWindow( ( WindowPtr )theFrontBuffer );
    DSpContext_SetState( gDspContext, kDspContextState_Inactive );
    DSpContext_FadeGammaIn( NULL, NULL );
    DSpContext_Release( gDspContext );
    DSpShutdown();
}

```

Very little code to this function. We start out by calling DSpContext\_FadeGammaOut() to fade out the screen, next we dispose of the window. We call DSpContext\_SetState() to make our DrawSprocket context inactive, fade back in with DSpContext\_FadeGammaIn(). We call DSpContext\_Release() to unregister our application from DrawSprocket, this will mean that our application can't use DrawSprocket anymore. Finally, we call DSpShutdown() to quit the process of quitting.

### The SetupAGL() Function

```

/*****> SetupAGL() <*****/
AGLContext SetupAGL( AGLDrawable window )
{
    GLint attrib[] = { AGL_RGBA, AGL_DEPTH_SIZE, 24, AGL_DOUBLEBUFFER, AGL_NO_SRGB };
    AGLPixelFormat format;
    AGLContext context;
    GLboolean ok;

    // Choose an rgb pixel format
    format = aglChoosePixelFormat( NULL, 0, attrib );
    if ( format == NULL )
        return NULL;
}

```

To create a pixel format we pass `aglChoosePixelFormat()` our attributes (much like we did when creating our DrawSprocket

```
// Create an AGL context
context = aglCreateContext( format, NULL );
if ( context == NULL )
    return NULL;
```

We use `aglCreateContext()` to create our AGL context from our pixel format that we just chose.

```
// Attach the window to the context
ok = aglSetDrawable( context, window );
if ( !ok )
    return NULL;
```

`aglSetDrawable()` sets a drawable (in this case a window) for the context. With a drawable set when you use OpenGL to r

```
// Make the context the current context
ok = aglSetCurrentContext( context );
if ( !ok )
    return NULL;
```

And then you have to set the AGL context that we made as the current context. Notice that all along we've been making : used 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 j

### The CleanupAGL() Function

```
/******> CleanupAGL() <*****/
void CleanupAGL( AGLContext context )
{
    aglSetCurrentContext( NULL );
    aglSetDrawable( context, NULL );
    aglDestroyContext( context );
}
```

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

```
/******> Reshape3D() <*****/
void Reshape3D( int w, int h )
{
    glViewport( 0, 0, w, h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, ( GLfloat )w / ( GLfloat )h, 0.5, 50 );
    glMatrixMode( GL_MODELVIEW );
}
```

```

    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 (glLoadIdentity(), glTranslatef(), gluPerspective()).

### The InitGL() Function

```

/*****> InitGL() <*****/
void InitGL( void )
{
    // Load some textures
    LoadGLTexture( "Tutorial.sgi", &gTutorialTexture, LINEAR );
    glShadeModel( GL_SMOOTH );
    glEnable( GL_TEXTURE_2D );
}

```

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 you a quick overview of what it does. Loading the actual image, it leaves that up to tkRGBImageLoad(), or whatever image loader I am using at the time, but it does set the texture filter to 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 in front or behind each other, so this code just sets a depth test function and enables it.

```

// 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 );
glLightModel( 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 and color 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;

```

I just continue adding to a rotation (by the way, all this simple application does is rotate a rectangle which has a differen

```

// 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 buf

```

// 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 `glutSwapBuffers()` that we set for our 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 );

```

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_NEAREST );
        glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
        glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, tempTexture->sizeX, tempTexture->sizeY, 0, GL_RGB, GL_NEAREST );
        break;
    case LINEAR:
        // Create Linear Filtered Texture
        glBindTexture( GL_TEXTURE_2D, *texture );
        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_RGB, tempTexture->sizeX, tempTexture->sizeY, 0, GL_RGB, GL_LINEAR );
        break;
    case MIPMAP:
        // Create MipMapped Texture
        glBindTexture( GL_TEXTURE_2D, *texture );
        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, tempTexture->sizeX, tempTexture->sizeY, GL_RGB, GL_LINEAR );
        break;
}

```

Each case in the switch statement takes `tempTexture` and sets a filter (`GL_NEAREST` or `GL_LINEAR`) and then calls `glTexImage2D()` or `gluBuild2DMipmaps()`, which is more than just a filter, it actually 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 glDelete() because if you don't they will stay in your 3D graphics accelerator card's memory until the computer is shut down, another time.

### **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](mailto:classic@sover.net). I hope you found this tutorial usefu

The OpenGL<sup>®</sup> 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
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
<b>2</b>	<b>OpenGL Operation</b>	<b>4</b>
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	Begin/End Paradigm . . . . .	12
2.6.1	Begin and End Objects . . . . .	15
2.6.2	Polygon Edges . . . . .	18
2.6.3	GL Commands within <b>Begin/End</b> . . . . .	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

2.13.1	Lighting . . . . .	44
2.13.2	Lighting Parameter Specification . . . . .	49
2.13.3	<b>ColorMaterial</b> . . . . .	51
2.13.4	Lighting State . . . . .	53
2.13.5	Color Index Lighting . . . . .	53
2.13.6	Clamping or Masking . . . . .	54
2.13.7	Flatshading . . . . .	54
2.13.8	Color and Texture Coordinate Clipping . . . . .	55
2.13.9	Final Color Processing . . . . .	56
<b>3</b>	<b>Rasterization</b> . . . . .	<b>57</b>
3.1	Invariance . . . . .	59
3.2	Antialiasing . . . . .	59
3.3	Points . . . . .	60
3.3.1	Point Rasterization State . . . . .	62
3.4	Line Segments . . . . .	62
3.4.1	Basic Line Segment Rasterization . . . . .	64
3.4.2	Other Line Segment Features . . . . .	66
3.4.3	Line Rasterization State . . . . .	69
3.5	Polygons . . . . .	70
3.5.1	Basic Polygon Rasterization . . . . .	70
3.5.2	Stippling . . . . .	72
3.5.3	Antialiasing . . . . .	72
3.5.4	Options Controlling Polygon Rasterization . . . . .	73
3.5.5	Depth Offset . . . . .	73
3.5.6	Polygon Rasterization State . . . . .	75
3.6	Pixel Rectangles . . . . .	75
3.6.1	Pixel Storage Modes . . . . .	75
3.6.2	The Imaging Subset . . . . .	76
3.6.3	Pixel Transfer Modes . . . . .	78
3.6.4	Rasterization of Pixel Rectangles . . . . .	88
3.6.5	Pixel Transfer Operations . . . . .	100
3.7	Bitmaps . . . . .	110
3.8	Texturing . . . . .	111
3.8.1	Texture Image Specification . . . . .	112
3.8.2	Alternate Texture Image Specification Commands . . . . .	118
3.8.3	Texture Parameters . . . . .	123
3.8.4	Texture Wrap Modes . . . . .	124
3.8.5	Texture Minification . . . . .	125
3.8.6	Texture Magnification . . . . .	131

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 Sum . . . . .	138
3.10	Fog . . . . .	138
3.11	Antialiasing Application . . . . .	140
<b>4</b>	<b>Per-Fragment Operations and the Framebuffer</b>	<b>141</b>
4.1	Per-Fragment Operations . . . . .	142
4.1.1	Pixel Ownership Test . . . . .	142
4.1.2	Scissor test . . . . .	143
4.1.3	Alpha test . . . . .	143
4.1.4	Stencil test . . . . .	144
4.1.5	Depth buffer test . . . . .	145
4.1.6	Blending . . . . .	146
4.1.7	Dithering . . . . .	149
4.1.8	Logical Operation . . . . .	150
4.2	Whole Framebuffer Operations . . . . .	150
4.2.1	Selecting a Buffer for Writing . . . . .	150
4.2.2	Fine Control of Buffer Updates . . . . .	152
4.2.3	Clearing the Buffers . . . . .	153
4.2.4	The Accumulation Buffer . . . . .	155
4.3	Drawing, 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 . . . . .	162
<b>5</b>	<b>Special Functions</b>	<b>164</b>
5.1	Evaluators . . . . .	164
5.2	Selection . . . . .	170
5.3	Feedback . . . . .	173
5.4	Display Lists . . . . .	175
5.5	Flush and Finish . . . . .	179
5.6	Hints . . . . .	179
<b>6</b>	<b>State and State Requests</b>	<b>181</b>
6.1	Querying GL State . . . . .	181
6.1.1	Simple Queries . . . . .	181

6.1.2	Data Conversions . . . . .	182
6.1.3	Enumerated Queries . . . . .	182
6.1.4	Texture Queries . . . . .	184
6.1.5	Stipple Query . . . . .	185
6.1.6	Color Matrix Query . . . . .	185
6.1.7	Color Table Query . . . . .	185
6.1.8	Convolution Query . . . . .	186
6.1.9	Histogram Query . . . . .	187
6.1.10	Minmax Query . . . . .	188
6.1.11	Pointer and String Queries . . . . .	189
6.1.12	Saving and Restoring State . . . . .	189
6.2	State Tables . . . . .	193
<b>A</b>	<b>Invariance</b>	<b>218</b>
A.1	Repeatability . . . . .	218
A.2	Multi-pass Algorithms . . . . .	219
A.3	Invariance Rules . . . . .	219
A.4	What All This Means . . . . .	221
<b>B</b>	<b>Corollaries</b>	<b>222</b>
<b>C</b>	<b>Version 1.1</b>	<b>225</b>
C.1	Vertex Array . . . . .	225
C.2	Polygon Offset . . . . .	226
C.3	Logical Operation . . . . .	226
C.4	Texture Image Formats . . . . .	226
C.5	Texture Replace Environment . . . . .	226
C.6	Texture Proxies . . . . .	227
C.7	Copy Texture and Subtexture . . . . .	227
C.8	Texture Objects . . . . .	227
C.9	Other Changes . . . . .	227
C.10	Acknowledgements . . . . .	228
<b>D</b>	<b>Version 1.2</b>	<b>230</b>
D.1	Three-Dimensional Texturing . . . . .	230
D.2	BGRA Pixel Formats . . . . .	230
D.3	Packed Pixel Formats . . . . .	230
D.4	Normal Rescaling . . . . .	231
D.5	Separate Specular Color . . . . .	231
D.6	Texture Coordinate Edge Clamping . . . . .	231

D.7	Texture Level of Detail Control . . . . .	232
D.8	Vertex Array Draw Element Range . . . . .	232
D.9	Imaging Subset . . . . .	232
D.9.1	Color Tables . . . . .	232
D.9.2	Convolution . . . . .	233
D.9.3	Color Matrix . . . . .	233
D.9.4	Pixel Pipeline Statistics . . . . .	234
D.9.5	Constant Blend Color . . . . .	234
D.9.6	New Blending Equations . . . . .	234
D.10	Acknowledgements . . . . .	234
<b>E</b>	<b>Version 1.2.1</b>	<b>238</b>
<b>F</b>	<b>ARB Extensions</b>	<b>239</b>
F.1	Naming Conventions . . . . .	239
F.2	Multitexture . . . . .	240
F.2.1	Dependencies . . . . .	240
F.2.2	Issues . . . . .	240
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) . . . . .	244
F.2.7	Changes to Section 2.10.4 (Generating Texture Coordinates) . . . . .	245
F.2.8	Changes to Section 2.12 (Current Raster Position) . . . . .	246
F.2.9	Changes to Section 3.8 (Texturing) . . . . .	246
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) . . . . .	249
F.2.14	Changes to Section 5.3 (Feedback) . . . . .	249
F.2.15	Changes to Section 6.1.2 (Data Conversions) . . . . .	251
F.2.16	Changes to Section 6.1.12 (Saving and Restoring State)	251
	<b>Index of OpenGL Commands</b>	<b>256</b>

# List of Figures

2.1	Block diagram of the GL. . . . .	9
2.2	Creation of a processed vertex from a transformed vertex and current values. . . . .	13
2.3	Primitive assembly and processing. . . . .	13
2.4	Triangle strips, fans, and independent triangles. . . . .	16
2.5	Quadrilateral strips and independent quadrilaterals. . . . .	17
2.6	Vertex transformation sequence. . . . .	28
2.7	Current raster position. . . . .	41
2.8	Processing of RGBA colors. . . . .	43
2.9	Processing of color indices. . . . .	43
2.10	ColorMaterial operation. . . . .	51
3.1	Rasterization. . . . .	57
3.2	Rasterization of non-antialiased wide points. . . . .	61
3.3	Rasterization of antialiased wide points. . . . .	61
3.4	Visualization of Bresenham's algorithm. . . . .	64
3.5	Rasterization of non-antialiased wide lines. . . . .	67
3.6	The region used in rasterizing an antialiased line segment. . . . .	69
3.7	Operation of <b>DrawPixels</b> . . . . .	88
3.8	Selecting a subimage from an image . . . . .	93
3.9	A bitmap and its associated parameters. . . . .	110
3.10	A texture image and the coordinates used to access it. . . . .	118
4.1	Per-fragment operations. . . . .	142
4.2	Operation of <b>ReadPixels</b> . . . . .	156
4.3	Operation of <b>CopyPixels</b> . . . . .	162
5.1	Map Evaluation. . . . .	166
5.2	Feedback syntax. . . . .	176

F.1	Creation of a processed vertex from a transformed vertex and current values. . . . .	241
F.2	Current raster position. . . . .	246
F.3	Multitexture pipeline. . . . .	249

# 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 <b>InterleavedArrays</b> . . . . .	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	<b>PixelStore</b> parameters pertaining to one or more of <b>DrawPixels</b> , <b>TexImage1D</b> , <b>TexImage2D</b> , and <b>TexImage3D</b> . . . . .	76
3.2	<b>PixelTransfer</b> parameters. . . . .	78
3.3	<b>PixelMap</b> parameters. . . . .	79
3.4	Color table names. . . . .	80
3.5	<b>DrawPixels</b> and <b>ReadPixels</b> types . . . . .	91
3.6	<b>DrawPixels</b> and <b>ReadPixels</b> formats. . . . .	92
3.7	Swap Bytes Bit ordering. . . . .	92
3.8	Packed pixel formats. . . . .	94
3.9	<b>UNSIGNED_BYTE</b> formats. Bit numbers are indicated for each component. . . . .	95
3.10	<b>UNSIGNED_SHORT</b> formats . . . . .	96
3.11	<b>UNSIGNED_INT</b> formats . . . . .	97
3.12	Packed pixel field assignments . . . . .	98
3.13	Color table lookup. . . . .	103
3.14	Computation of filtered color components. . . . .	104
3.15	Conversion from RGBA pixel components to internal texture, table, or filter components. . . . .	114
3.16	Correspondence of sized internal formats to base internal formats. . . . .	115

3.17	Texture parameters and their values. . . . .	124
3.18	Replace and modulate texture functions. . . . .	136
3.19	Decal and blend texture functions. . . . .	137
4.1	Values controlling the source blending function and the source blending values they compute. $f = \min(A_s, 1 - A_d)$ . . . . .	148
4.2	Values controlling the destination blending function and the destination blending values they compute. . . . .	148
4.3	Arguments to <b>LogicOp</b> and their corresponding operations. . . . .	151
4.4	Arguments to <b>DrawBuffer</b> and the buffers that they indicate. . . . .	152
4.5	<b>PixelStore</b> parameters pertaining to <b>ReadPixels</b> , <b>GetTexImage1D</b> , <b>GetTexImage2D</b> , <b>GetTexImage3D</b> , <b>GetColorTable</b> , <b>GetConvolutionFilter</b> , <b>GetSeparable-</b> <b>Filter</b> , <b>GetHistogram</b> , and <b>GetMinmax</b> . . . . .	158
4.6	<b>ReadPixels</b> index masks. . . . .	160
4.7	<b>ReadPixels</b> GL Data Types and Reversed component con- version formulas. . . . .	161
5.1	Values specified by the <i>target</i> to <b>Map1</b> . . . . .	165
5.2	Correspondence of feedback type to number of values per vertex. . . . .	174
6.1	Texture, table, and filter return values. . . . .	185
6.2	Attribute groups . . . . .	191
6.3	State variable types . . . . .	192
6.4	GL Internal begin-end state variables (inaccessible) . . . . .	194
6.5	Current Values and Associated Data . . . . .	195
6.6	Vertex Array Data . . . . .	196
6.7	Transformation state . . . . .	197
6.8	Coloring . . . . .	198
6.9	Lighting (see also Table 2.7 for defaults) . . . . .	199
6.10	Lighting (cont.) . . . . .	200
6.11	Rasterization . . . . .	201
6.12	Texture Objects . . . . .	202
6.13	Texture Objects (cont.) . . . . .	203
6.14	Texture Environment and Generation . . . . .	204
6.15	Pixel Operations . . . . .	205
6.16	Framebuffer Control . . . . .	206
6.17	Pixels . . . . .	207
6.18	Pixels (cont.) . . . . .	208
6.19	Pixels (cont.) . . . . .	209

6.20	Pixels (cont.) . . . . .	210
6.21	Pixels (cont.) . . . . .	211
6.22	Evaluators ( <b>GetMap</b> takes a map name) . . . . .	212
6.23	Hints . . . . .	213
6.24	Implementation Dependent Values . . . . .	214
6.25	More Implementation Dependent Values . . . . .	215
6.26	Implementation Dependent Pixel Depths . . . . .	216
6.27	Miscellaneous . . . . .	217
F.1	Changes to State Tables . . . . .	252
F.2	Changes to State Tables (cont.) . . . . .	253
F.3	New State Introduced by Multitexture . . . . .	254
F.4	New Implementation-Dependent Values Introduced by Multitexture . . . . .	255

# 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 frame-buffer. 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 two-dimensional 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 form<sup>1</sup>

---

<sup>1</sup>The declarations shown in this document apply to ANSI C. Languages such as C++

Letter	Corresponding GL Type
<b>b</b>	byte
<b>s</b>	short
<b>i</b>	int
<b>f</b>	float
<b>d</b>	double
<b>ub</b>	ubyte
<b>us</b>	ushort
<b>ui</b>	uint

Table 2.1: Correspondence of command suffix letters to GL argument types. Refer to Table 2.2 for definitions of the GL types.

$$rtype \text{ Name}\{\epsilon 1234\}\{\epsilon \mathbf{b} \mathbf{s} \mathbf{i} \mathbf{f} \mathbf{d} \mathbf{ub} \mathbf{us} \mathbf{ui}\}\{\epsilon \mathbf{v}\} \\ ( [args, ] T arg1, \dots, T argN [ , args] );$$

*rtype* is the return type of the function. The braces ( $\{\}$ ) enclose a series of characters (or character pairs) of which one is selected.  $\epsilon$  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
<code>boolean</code>	1	Boolean
<code>byte</code>	8	signed 2's complement binary integer
<code>ubyte</code>	8	unsigned binary integer
<code>short</code>	16	signed 2's complement binary integer
<code>ushort</code>	16	unsigned binary integer
<code>int</code>	32	signed 2's complement binary integer
<code>uint</code>	32	unsigned binary integer
<code>sizei</code>	32	Non-negative binary integer size
<code>enum</code>	32	Enumerated binary integer value
<code>bitfield</code>	32	Bit field
<code>float</code>	32	Floating-point value
<code>clampf</code>	32	Floating-point value clamped to $[0, 1]$
<code>double</code>	64	Floating-point value
<code>clampd</code>	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.

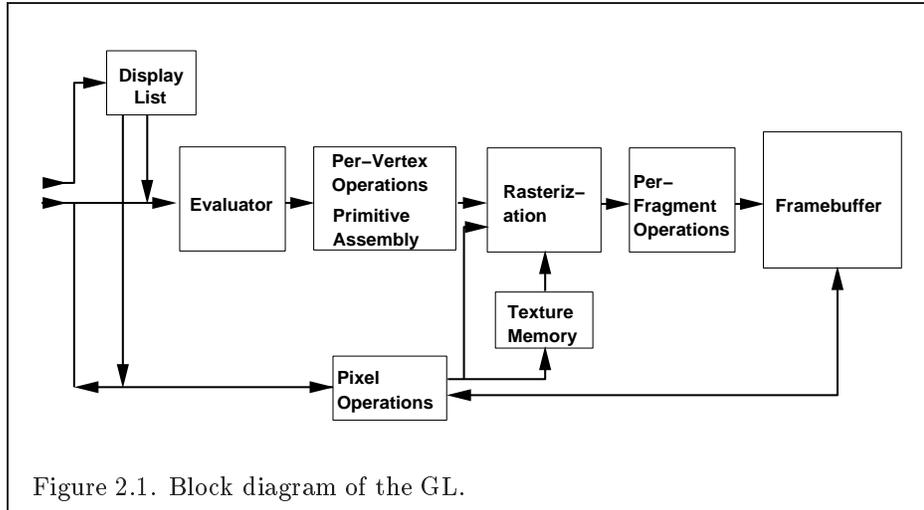


Figure 2.1. Block diagram of the GL.

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 flag-code 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 command ignored?
INVALID_ENUM	enum argument out of range	Yes
INVALID_VALUE	Numeric argument out of range	Yes
INVALID_OPERATION	Operation illegal in current state	Yes
STACK_OVERFLOW	Command would cause a stack overflow	Yes
STACK_UNDERFLOW	Command would cause a stack underflow	Yes
OUT_OF_MEMORY	Not enough memory left to execute command	Unknown
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-

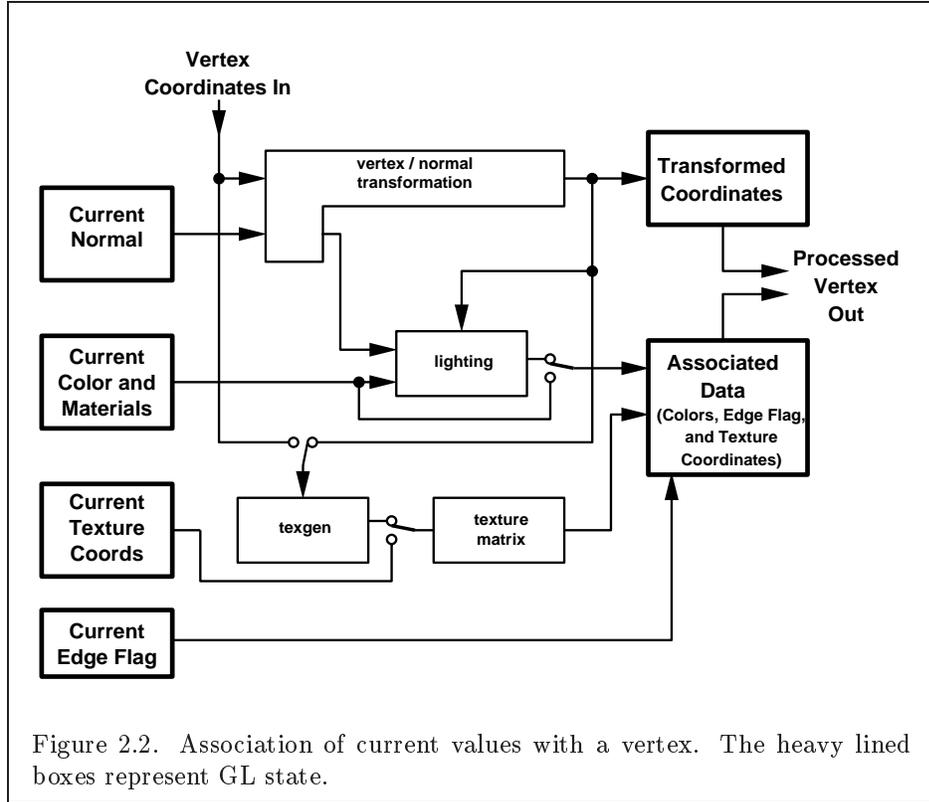


Figure 2.2. Association of current values with a vertex. The heavy lined boxes represent GL state.

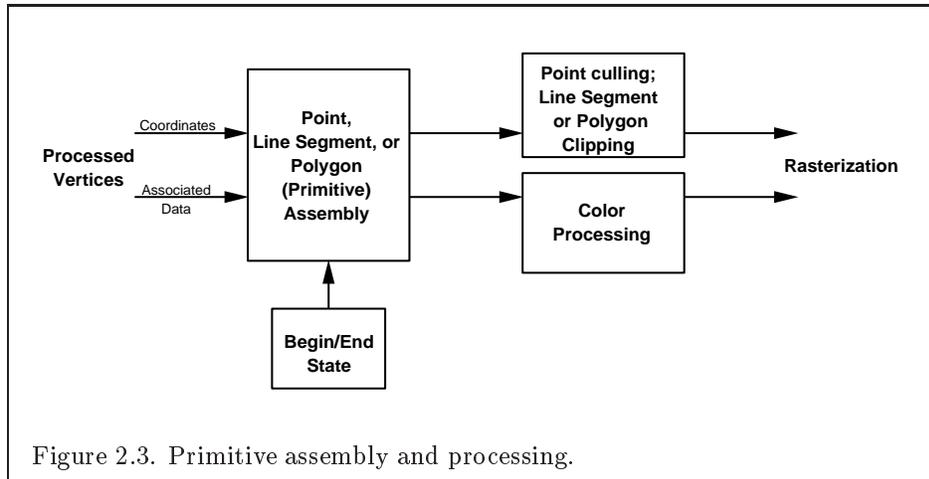


Figure 2.3. Primitive assembly and processing.

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  $i - 1$ st. 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.

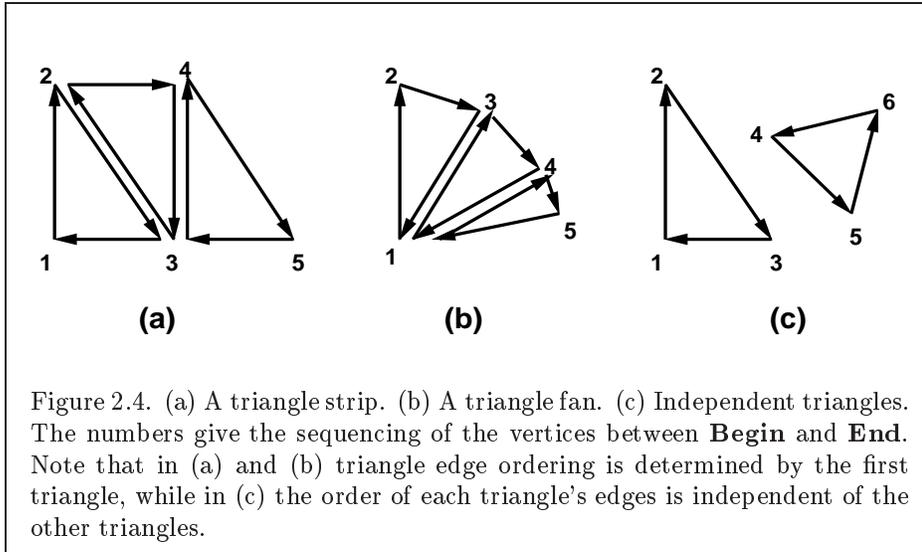
Only convex polygons are guaranteed to be drawn correctly by the GL. If a specified polygon is nonconvex when projected onto the window, then the rendered polygon need only lie within the convex hull of the projected vertices defining its boundary.

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

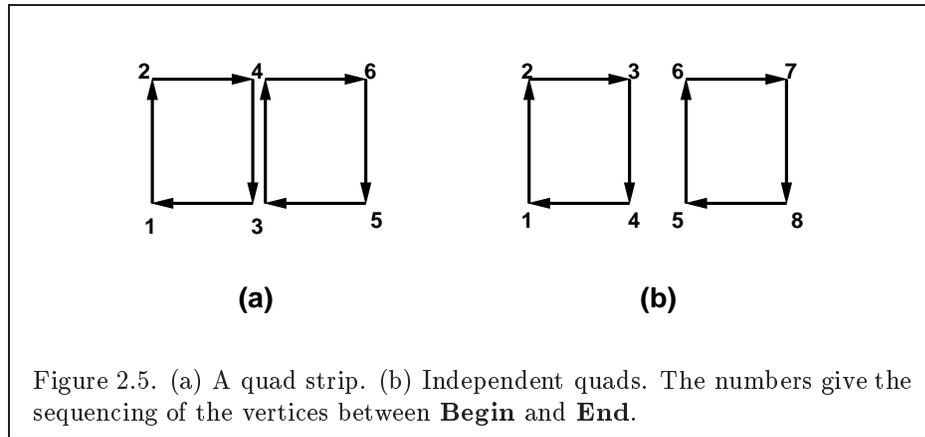


exception: each vertex after the first always replaces vertex B of the two stored vertices. The vertices of a triangle fan are enclosed between **Begin** and **End** when the value of the argument to **Begin** is **TRIANGLE\_FAN**.

**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  $3i + 1$ st,  $3i + 2$ nd, and  $3i + 3$ rd vertices (in that order) determine a triangle for each  $i = 0, 1, \dots, n - 1$ , 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 edge-sharing 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  $v_1, \dots, v_m$ , 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  $i = 0, \dots, \lfloor m/2 \rfloor$ . 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  $4j + 1$ st, the  $4j + 2$ nd, the  $4j + 3$ rd, and the  $4j + 4$ th, generate a single quad, for  $j = 0, 1, \dots, n - 1$ . The total number of vertices between **Begin** and **End** is  $4n + k$ , where  $0 \leq k \leq 3$ ; 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**, **TexCoord**), 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**, **EdgeFlagPointer**, **TexCoordPointer**, **ColorPointer**, **IndexPointer**, **NormalPointer**, **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 floating-point 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  $(R, G, B, A) = (1, 1, 1, 1)$ . 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
<b>VertexPointer</b>	2,3,4	short, int, float, double
<b>NormalPointer</b>	3	byte, short, int, float, double
<b>ColorPointer</b>	3,4	byte, ubyte, short, ushort, int, uint, float, double
<b>IndexPointer</b>	1	ubyte, short, int, float, double
<b>TexCoordPointer</b>	1,2,3,4	short, int, float, double
<b>EdgeFlagPointer</b>	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, size_t 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();
}

```

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();
}

```

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  $end - start + 1$  is 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

<i>format</i>	<i>e<sub>t</sub></i>	<i>e<sub>c</sub></i>	<i>e<sub>n</sub></i>	<i>s<sub>t</sub></i>	<i>s<sub>c</sub></i>	<i>s<sub>v</sub></i>	<i>t<sub>c</sub></i>
V2F	<i>False</i>	<i>False</i>	<i>False</i>			2	
V3F	<i>False</i>	<i>False</i>	<i>False</i>			3	
C4UB_V2F	<i>False</i>	<i>True</i>	<i>False</i>		4	2	UNSIGNED_BYTE
C4UB_V3F	<i>False</i>	<i>True</i>	<i>False</i>		4	3	UNSIGNED_BYTE
C3F_V3F	<i>False</i>	<i>True</i>	<i>False</i>		3	3	FLOAT
N3F_V3F	<i>False</i>	<i>False</i>	<i>True</i>			3	
C4F_N3F_V3F	<i>False</i>	<i>True</i>	<i>True</i>		4	3	FLOAT
T2F_V3F	<i>True</i>	<i>False</i>	<i>False</i>	2		3	
T4F_V4F	<i>True</i>	<i>False</i>	<i>False</i>	4		4	
T2F_C4UB_V3F	<i>True</i>	<i>True</i>	<i>False</i>	2	4	3	UNSIGNED_BYTE
T2F_C3F_V3F	<i>True</i>	<i>True</i>	<i>False</i>	2	3	3	FLOAT
T2F_N3F_V3F	<i>True</i>	<i>False</i>	<i>True</i>	2		3	
T2F_C4F_N3F_V3F	<i>True</i>	<i>True</i>	<i>True</i>	2	4	3	FLOAT
T4F_C4F_N3F_V4F	<i>True</i>	<i>True</i>	<i>True</i>	4	4	4	FLOAT

<i>format</i>	<i>p<sub>c</sub></i>	<i>p<sub>n</sub></i>	<i>p<sub>v</sub></i>	<i>s</i>
V2F			0	2 <i>f</i>
V3F			0	3 <i>f</i>
C4UB_V2F	0		<i>c</i>	<i>c</i> + 2 <i>f</i>
C4UB_V3F	0		<i>c</i>	<i>c</i> + 3 <i>f</i>
C3F_V3F	0		3 <i>f</i>	6 <i>f</i>
N3F_V3F		0	3 <i>f</i>	6 <i>f</i>
C4F_N3F_V3F	0	4 <i>f</i>	7 <i>f</i>	10 <i>f</i>
T2F_V3F			2 <i>f</i>	5 <i>f</i>
T4F_V4F			4 <i>f</i>	8 <i>f</i>
T2F_C4UB_V3F	2 <i>f</i>		<i>c</i> + 2 <i>f</i>	<i>c</i> + 5 <i>f</i>
T2F_C3F_V3F	2 <i>f</i>		5 <i>f</i>	8 <i>f</i>
T2F_N3F_V3F		2 <i>f</i>	5 <i>f</i>	8 <i>f</i>
T2F_C4F_N3F_V3F	2 <i>f</i>	6 <i>f</i>	9 <i>f</i>	12 <i>f</i>
T4F_C4F_N3F_V4F	4 <i>f</i>	8 <i>f</i>	11 <i>f</i>	15 <i>f</i>

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( $s_c, t_c, \mathbf{str}, \mathit{pointer} + p_c$ );
    } else {
        DisableClientState(COLOR_ARRAY);
    }
    if ( $e_n$ ) {
        EnableClientState(NORMAL_ARRAY);
        NormalPointer(FLOAT, str, pointer +  $p_n$ );
    } else {
        DisableClientState(NORMAL_ARRAY);
    }
    EnableClientState(VERTEX_ARRAY);
    VertexPointer( $s_v$ , FLOAT, 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 (x1, y1, x2, y2);
```

is exactly the same as the following sequence of commands:

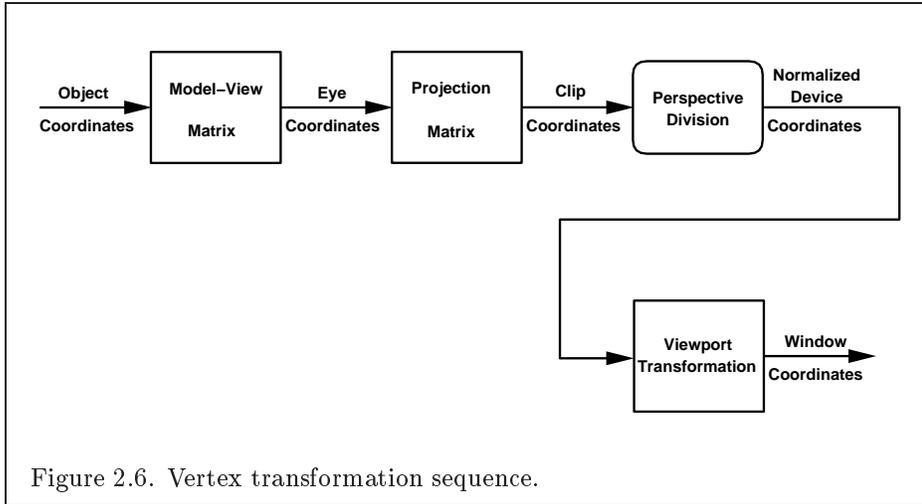
```
Begin(POLYGON);
  Vertex2(x1, y1);
  Vertex2(x2, y1);
  Vertex2(x2, y2);
  Vertex2(x1, y2);
End();
```

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 four-dimensional, consisting of  $x$ ,  $y$ ,  $z$ , and  $w$  coordinates (in that order). The model-view and perspective matrices are thus  $4 \times 4$ .

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  $P$  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

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{pmatrix}.$$

### 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  $k \in \{0, 1, \dots, 2^m - 1\}$ , 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 lower-left 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 `COLOR` is 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  $4 \times 4$  matrix stored in column-major order as 16 consecutive floating-point values, i.e. as

$$\begin{pmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{pmatrix}.$$

(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. **Rotate**, **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  $z$  );
```

$\theta$  gives an angle of rotation in degrees; the coordinates of a vector  $\mathbf{v}$  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

$$\begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Let  $\mathbf{u} = \mathbf{v}/\|\mathbf{v}\| = (x' \ y' \ z')^T$ . If

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

then

$$R = \mathbf{u}\mathbf{u}^T + \cos \theta (I - \mathbf{u}\mathbf{u}^T) + \sin \theta S.$$

The arguments to

```
void Translate{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:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

```
void Scale{fd}( T x, T y, T z );
```

produces a general scaling along the  $x$ -,  $y$ -, and  $z$ - axes. The corresponding matrix is

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

For

```
void Frustum( double l, double r, double b, double t,
              double n, double f );
```

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

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

```
void Ortho( double l, double r, double b, double t,
            double n, double f );
```

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 lower-left 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  $n$  is 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  $4 \times 4$  matrix that is applied to texture coordinates. This matrix is applied as

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} \begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix},$$

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 four-valued integer indicating the current matrix mode, a stack of at least two  $4 \times 4$  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

```
void Enable( enum target );
```

and

```
void Disable( enum target );
```

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:

$$(n_x' \ n_y' \ n_z' \ q') = (n_x \ n_y \ n_z \ q) \cdot M^{-1}$$

where, if  $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$  are the associated vertex coordinates, then

$$q = \begin{cases} 0, & w = 0, \\ \frac{-(n_x \ n_y \ n_z) \begin{pmatrix} x \\ y \\ z \end{pmatrix}}{w}, & w \neq 0 \end{cases} \quad (2.1)$$

Implementations may choose instead to transform  $(n_x \ n_y \ n_z)$  to eye coordinates using

$$(n_x' \ n_y' \ n_z') = (n_x \ n_y \ n_z) \cdot M_u^{-1}$$

where  $M_u$  is the upper leftmost 3x3 matrix taken from  $M$ .

Rescale multiplies the transformed normals by a scale factor

$$(n_x'' \ n_y'' \ n_z'') = f(n_x' \ n_y' \ n_z')$$

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 model-view 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 poorly-conditioned (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

$$g = p_1x_o + p_2y_o + p_3z_o + p_4w_o.$$

$x_o$ ,  $y_o$ ,  $z_o$ , and  $w_o$  are the object coordinates of the vertex.  $p_1, \dots, p_4$  are specified by calling **TexGen** with *pname* set to `OBJECT_PLANE` in which case *params* points to an array containing  $p_1, \dots, p_4$ . 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

$$g = p'_1x_e + p'_2y_e + p'_3z_e + p'_4w_e$$

where

$$(p'_1 \ p'_2 \ p'_3 \ p'_4) = (p_1 \ p_2 \ p_3 \ p_4) M^{-1}$$

$x_e$ ,  $y_e$ ,  $z_e$ , and  $w_e$  are the eye coordinates of the vertex.  $p_1, \dots, p_4$  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  $p_1, \dots, p_4$  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  $\mathbf{u}$ . Denote the current normal, after transformation to eye coordinates, by  $\mathbf{n}'$ . Let  $\mathbf{r} = (r_x \ r_y \ r_z)^T$ , the reflection vector, be given by

$$\mathbf{r} = \mathbf{u} - 2\mathbf{n}'^T (\mathbf{n}'\mathbf{u}),$$

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 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**.

## 2.11 Clipping

Primitives are clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by

$$\begin{aligned} -w_c &\leq x_c \leq w_c \\ -w_c &\leq y_c \leq w_c \ . \\ -w_c &\leq z_c \leq w_c \end{aligned}$$

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

$$(p'_1 \ p'_2 \ p'_3 \ p'_4) = (p_1 \ p_2 \ p_3 \ p_4) M^{-1}$$

(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

$$(p'_1 \ p'_2 \ p'_3 \ p'_4) \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

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 $i$`  = `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,  $0 \leq t \leq 1$ , for each clipped vertex. If the coordinates of a clipped vertex are  $\mathbf{P}$  and the original vertices' coordinates are  $\mathbf{P}_1$  and  $\mathbf{P}_2$ , 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  $(p'_1 \ p'_2 \ p'_3 \ p'_4)$  (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  $(-p'_1 \ -p'_2 \ -p'_3 \ -p'_4)$  (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:

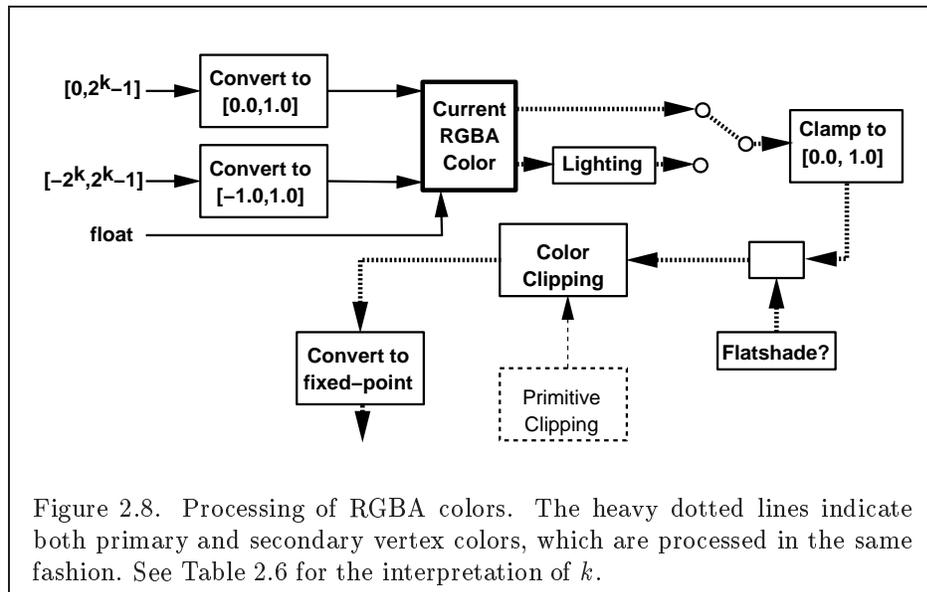
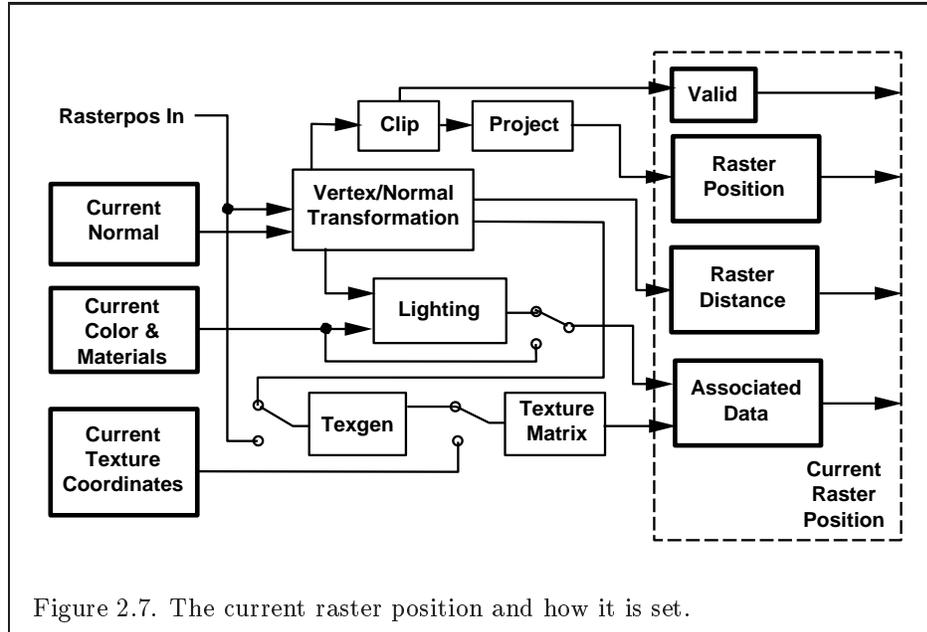
```
void RasterPos{234}{sifd}( T coords );
void RasterPos{234}{sifd}v( T coords );
```

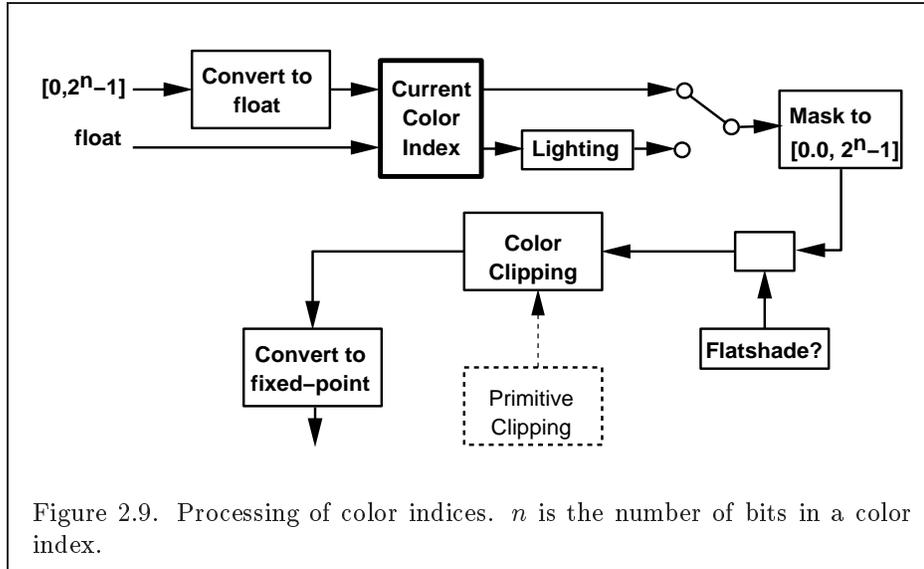
**RasterPos4** takes four values indicating  $x$ ,  $y$ ,  $z$ , and  $w$ . **RasterPos3** (or **RasterPos2**) is analogous, but sets only  $x$ ,  $y$ , and  $z$  with  $w$  implicitly set to 1 (or only  $x$  and  $y$  with  $z$  implicitly set to 0 and  $w$  implicitly set to 1).

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	$c$
double	$c$

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 direction given by  $x$ ,  $y$ , 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  $i = 0, \dots, n-1$ . ( $n$  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  $\overrightarrow{\mathbf{P}_1 \mathbf{P}_2}$  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  $\overrightarrow{\mathbf{P}_1 \mathbf{P}_2}$  is the unit vector corresponding to the direction specified by the  $x$ ,  $y$ , and  $z$  coordinates of  $\mathbf{P}_2$ ; if  $\mathbf{P}_1$  has a zero  $w$  coordinate and  $\mathbf{P}_2$  has a non-zero  $w$  coordinate then  $\overrightarrow{\mathbf{P}_1 \mathbf{P}_2}$  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  $\overrightarrow{\mathbf{P}_1 \mathbf{P}_2}$  is the unit vector obtained by normalizing the direction corresponding to  $\mathbf{P}_2 - \mathbf{P}_1$ .

If  $\mathbf{d}$  is an arbitrary direction, then let  $\hat{\mathbf{d}}$  be the unit vector in  $\mathbf{d}$ 's direction. Let  $\|\mathbf{P}_1 \mathbf{P}_2\|$  be the distance between  $\mathbf{P}_1$  and  $\mathbf{P}_2$ . Finally, let  $\mathbf{V}$  be the point corresponding to the vertex being lit, and  $\mathbf{n}$  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	(0.2, 0.2, 0.2, 1.0)	ambient color of material
$\mathbf{d}_{cm}$	color	(0.8, 0.8, 0.8, 1.0)	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 Parameters			
$\mathbf{a}_{cli}$	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 Model Parameters			
$\mathbf{a}_{cs}$	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 colors
$t_{bs}$	boolean	FALSE	use two-sided lighting mode

Table 2.7: Summary of lighting parameters. The range of individual color components is  $(-\infty, +\infty)$ .

color control,  $c_{es}$ . If  $c_{es} = \text{SINGLE\_COLOR}$ , then the equations to compute  $\mathbf{c}_{pri}$  and  $\mathbf{c}_{sec}$  are

$$\begin{aligned}\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\ &+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\ &+ \sum_{i=0}^{n-1} (att_i)(spot_i) [\mathbf{a}_{cm} * \mathbf{a}_{cli} \\ &\quad + (\mathbf{n} \odot \overline{\mathbf{VP}}_{pli}) \mathbf{d}_{cm} * \mathbf{d}_{cli} \\ &\quad + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}] \\ \mathbf{c}_{sec} &= (0, 0, 0, 0)\end{aligned}$$

If  $c_{es} = \text{SEPARATE\_SPECULAR\_COLOR}$ , then

$$\begin{aligned}\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\ &+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\ &+ \sum_{i=0}^{n-1} (att_i)(spot_i) [\mathbf{a}_{cm} * \mathbf{a}_{cli} \\ &\quad + (\mathbf{n} \odot \overline{\mathbf{VP}}_{pli}) \mathbf{d}_{cm} * \mathbf{d}_{cli}] \\ \mathbf{c}_{sec} &= \sum_{i=0}^{n-1} (att_i)(spot_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}\end{aligned}$$

where

$$f_i = \begin{cases} 1, & \mathbf{n} \odot \overline{\mathbf{VP}}_{pli} \neq 0, \\ 0, & \text{otherwise,} \end{cases} \quad (2.2)$$

$$\mathbf{h}_i = \begin{cases} \overline{\mathbf{VP}}_{pli} + \overline{\mathbf{VP}}_e, & v_{bs} = \text{TRUE}, \\ \overline{\mathbf{VP}}_{pli} + (0 \ 0 \ 1)^T, & v_{bs} = \text{FALSE}, \end{cases} \quad (2.3)$$

$$att_i = \begin{cases} \frac{1}{k_{0i} + k_{1i} \|\overline{\mathbf{VP}}_{pli}\| + k_{2i} \|\overline{\mathbf{VP}}_{pli}\|^2}, & \text{if } \mathbf{P}_{pli}'\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases} \quad (2.4)$$

$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases} \quad (2.5)$$

(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

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_w^i y_w^{i \oplus 1} - x_w^{i \oplus 1} y_w^i \quad (2.7)$$

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  $(i + 1) \bmod n$ . 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**.

### 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 `LIGHTi`, indicating that light *i* is to have the specified parameter set. The constants obey `LIGHTi = LIGHT0 + 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 “ $\infty$ ” 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
<b>Material Parameters (Material)</b>		
$\mathbf{a}_{cm}$	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
<b>Light Source Parameters (Light)</b>		
$\mathbf{a}_{cli}$	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
<b>Lighting Model Parameters (LightModel)</b>		
$\mathbf{a}_{cs}$	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 *i*th 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`.

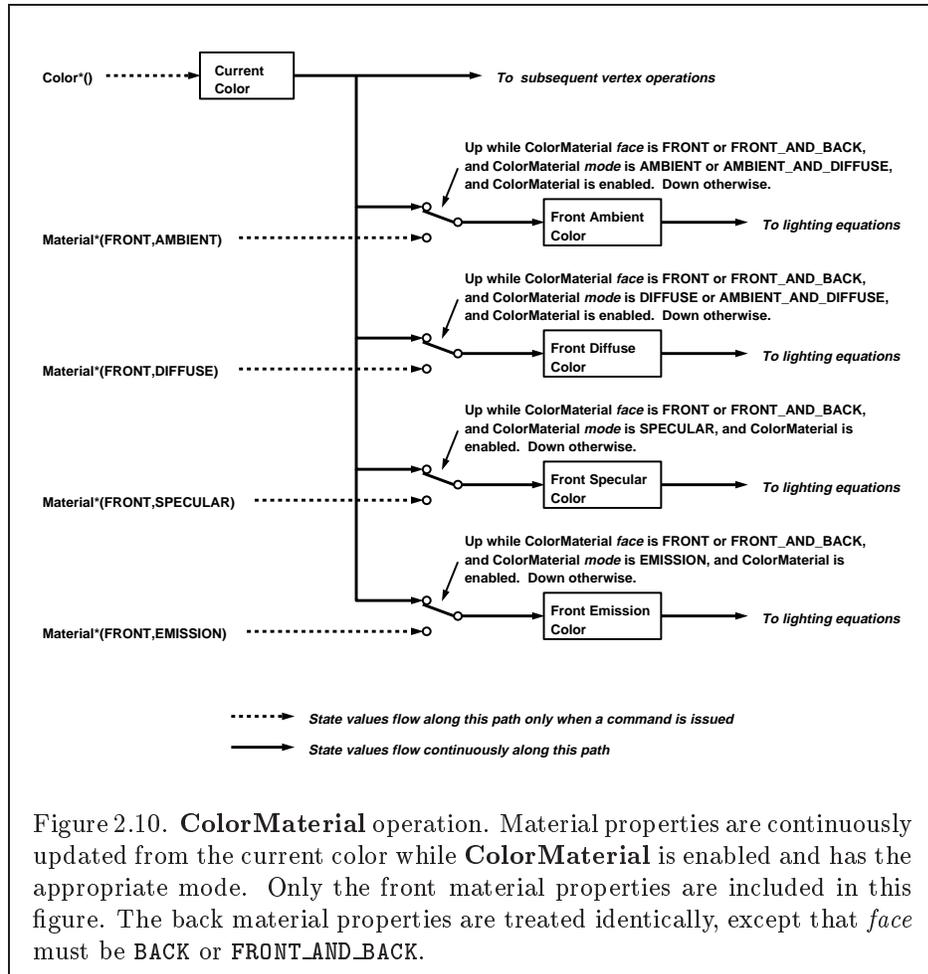
The command that controls which of these modes is selected is

```
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  $e_{cm}$ ,  $a_{cm}$ ,  $d_{cm}$  or  $s_{cm}$ , respectively, will track the current color. If *mode* is `AMBIENT_AND_DIFFUSE`, both  $a_{cm}$  and  $d_{cm}$  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  $a_{cm}$  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  $k \in \{0, 1, \dots, 2^m - 1\}$ , 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

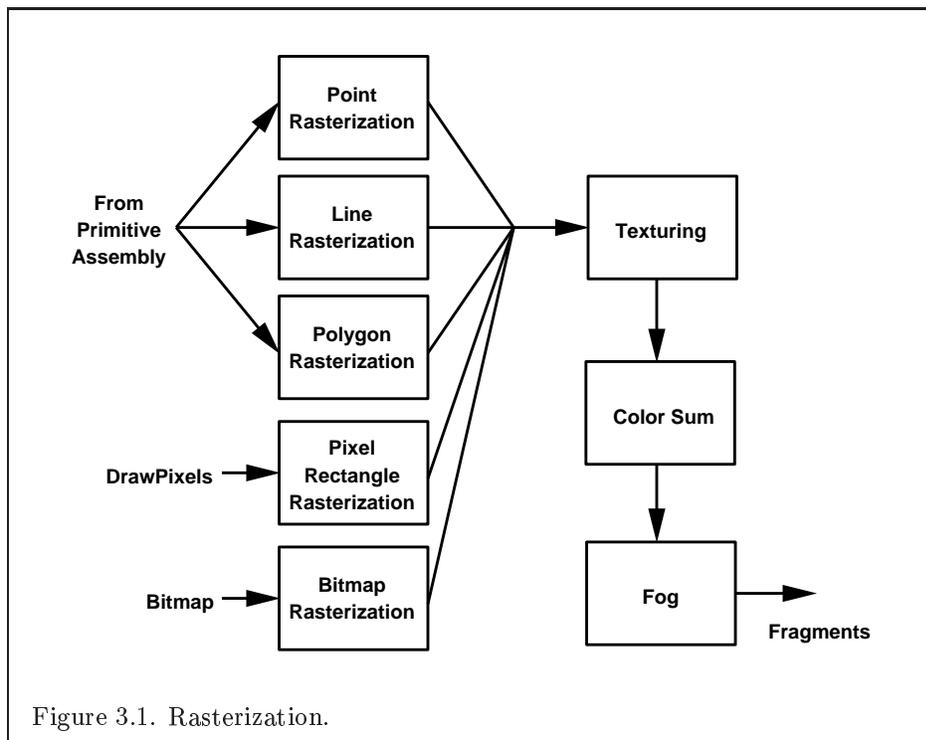
# Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional 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 lower-left 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 lower-left 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.

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 implementation-dependent 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

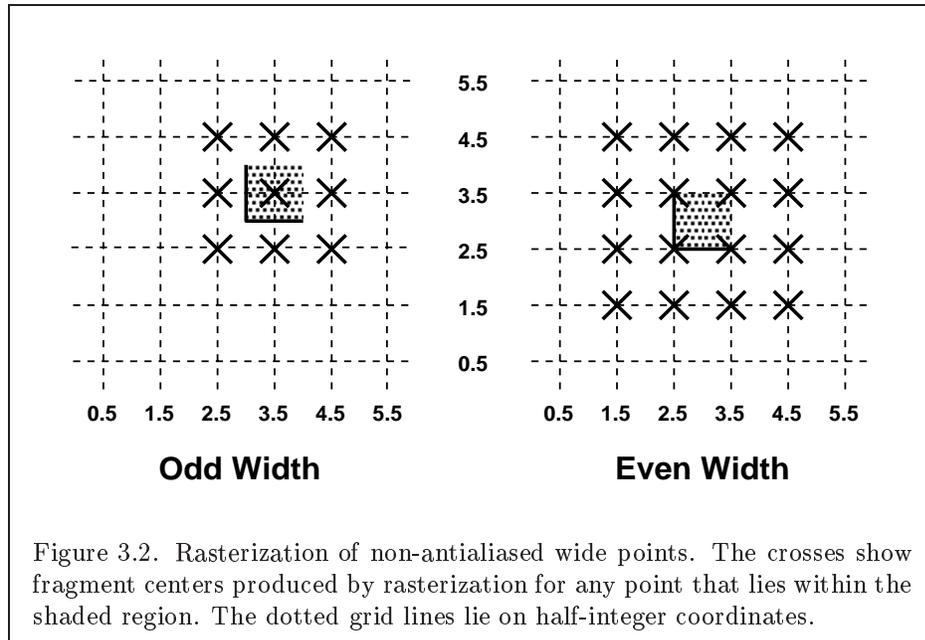
$$(x, y) = (\lfloor x_w + \frac{1}{2} \rfloor, \lfloor y_w + \frac{1}{2} \rfloor);$$

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, ..., 1.9, 2.0 are supported.

### 3.3.1 Point Rasterization State

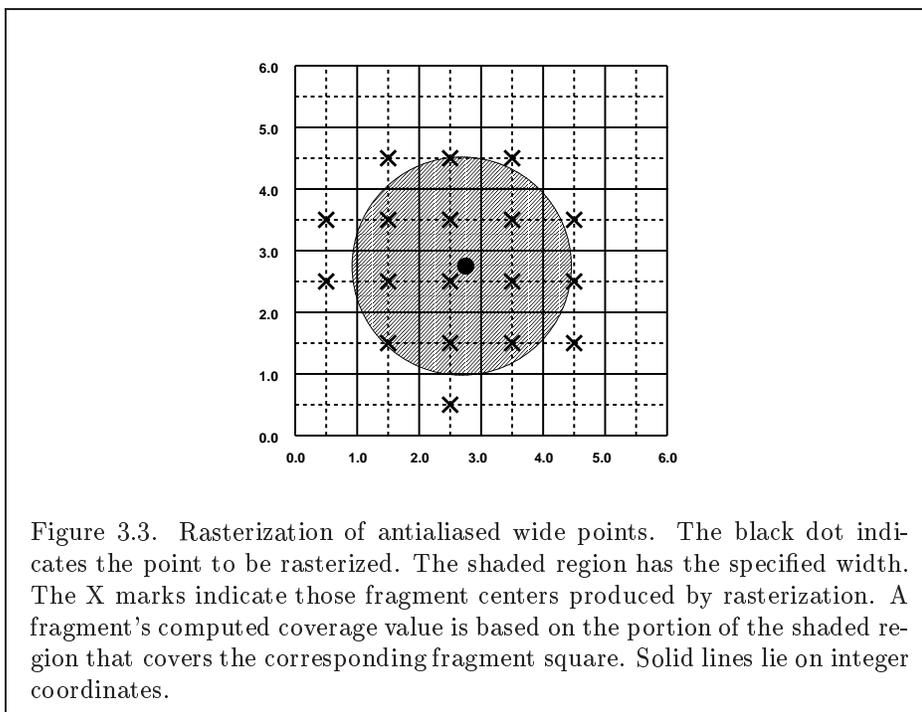
The state required to control point rasterization consists of the floating-point point width and a bit indicating whether or not antialiasing is enabled.

## 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



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:

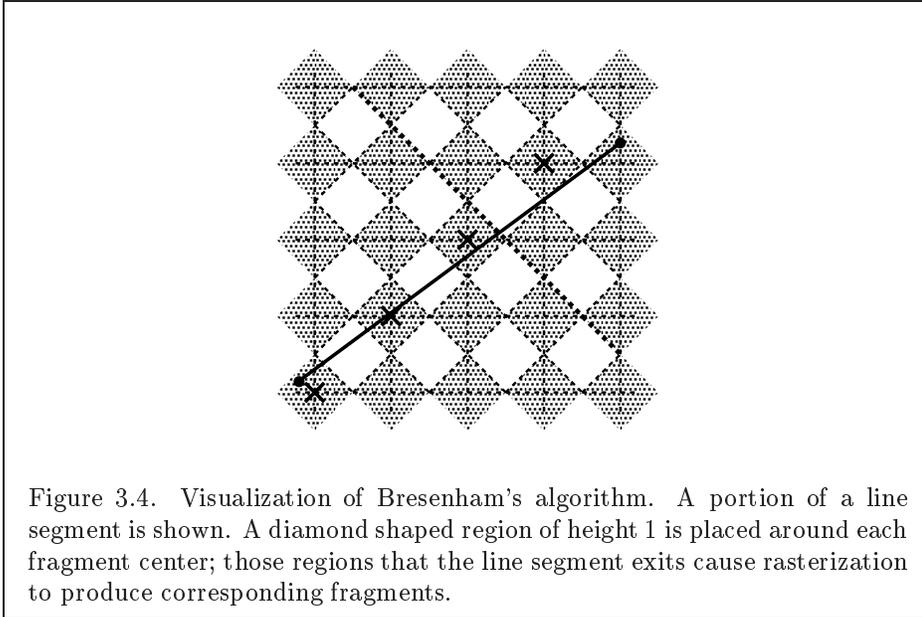
$$R_f = \{ (x, y) \mid |x - x_f| + |y - y_f| < 1/2. \}$$

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  $R_f$ .  $\epsilon$  is chosen to be so small that rasterizing the line segment produces the same fragments when  $\delta$  is substituted for  $\epsilon$  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:



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

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \quad (3.1)$$

(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

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)\alpha_a/w_a + t\alpha_b/w_b} \quad (3.2)$$

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  $\alpha_b = q_b$  ( $q_a$  and  $q_b$  are the homogeneous texture coordinates at the starting and ending endpoints of the segment; results are undefined if either of these is less than or equal to 0). Note that linear interpolation would use

$$f = (1-t)f_a/\alpha_a + tf_b/\alpha_b. \quad (3.3)$$

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

$$b = \lfloor s/r \rfloor \bmod 16,$$

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 implementation-dependent maximum non-antialiased line width. This implementation-dependent 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*<sub>0</sub>, *y*<sub>0</sub>) and (*x*<sub>1</sub>, *y*<sub>1</sub>) in window coordinates, the segment with endpoints (*x*<sub>0</sub>, *y*<sub>0</sub> - (*w* - 1)/2) and (*x*<sub>1</sub>, *y*<sub>1</sub> - (*w* - 1)/2) is rasterized, but

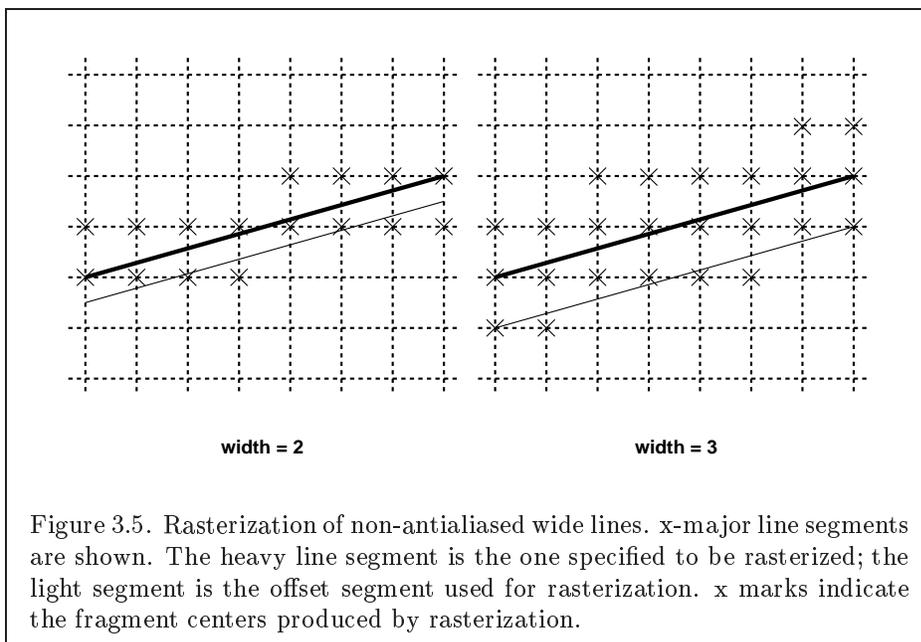


Figure 3.5. Rasterization of non-antialiased wide lines.  $x$ -major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization.  $x$  marks indicate the fragment centers produced by rasterization.

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-

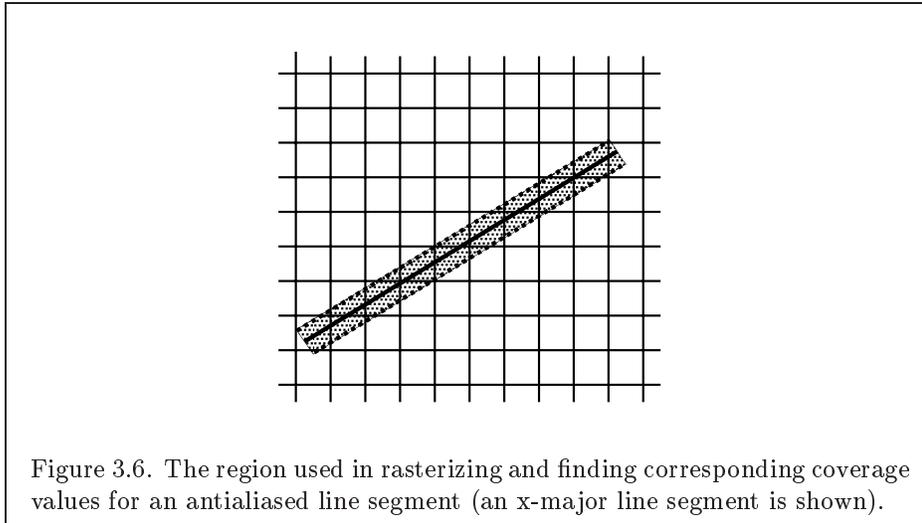


Figure 3.6. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

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.

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

$$a = \frac{A(pp_b p_c)}{A(p_a p_b p_c)}, \quad b = \frac{A(pp_a p_c)}{A(p_a p_b p_c)}, \quad c = \frac{A(pp_a p_b)}{A(p_a p_b p_c)},$$

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

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a\alpha_a/w_a + b\alpha_b/w_b + c\alpha_c/w_c} \quad (3.4)$$

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  $i$   $0 \leq a_i \leq 1$  and  $\sum_{i=1}^n a_i = 1$ . The values of the  $a_i$  may differ from fragment to fragment, but at vertex  $i$ ,  $a_j = 0, j \neq i$  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  $32 \times 32$  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 ( $x_w \bmod 32, y_w \bmod 32$ ) 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

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \quad (3.5)$$

where  $(x_w, y_w, z_w)$  is a point on the triangle.  $m$  may be approximated as

$$m = \max \left\{ \left| \frac{\partial z_w}{\partial x_w} \right|, \left| \frac{\partial z_w}{\partial y_w} \right| \right\}. \quad (3.6)$$

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

$$o = m * factor + r * units. \quad (3.7)$$

$m$  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 **DrawPixels** 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 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.
4. 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 **BlendEquation**, **BlendColor**, and the **BlendFunc** *modes* `CONSTANT_COLOR`, `ONE_MINUS_CONSTANT_COLOR`, `CONSTANT_ALPHA`, and `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	B	1	0.0
PIXEL_MAP_I_TO_A	color idx	A	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	B	B	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: single-precision 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 POST_CONVOLUTION_COLOR_TABLE POST_COLOR_MATRIX_COLOR_TABLE	regular
PROXY_COLOR_TABLE PROXY_POST_CONVOLUTION_COLOR_TABLE PROXY_POST_COLOR_MATRIX_COLOR_TABLE	proxy

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

```
void CopyColorTable( enum target, enum internalformat,
                    int x, int y, sizei width );
```

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 **CopyColorSubTable** 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 *N*th 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 **GetConvolutionParameteriv**, 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 **ConvolutionFilter2D** 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 **ConvolutionParameterfv** 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 `CopyPixels` with argument *type* set to `COLOR`, stopping after the final expansion to `RGBA`.

Subsequent processing is identical to that described for `ConvolutionFilter2D`, 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 **ConvolutionFilter1D**, 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 two-dimensional 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

```
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 accommodated 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

```
void Minmax( enum target, enum internalformat,
             boolean sink );
```

*target* must be `MINMAX`. *internalformat* 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.

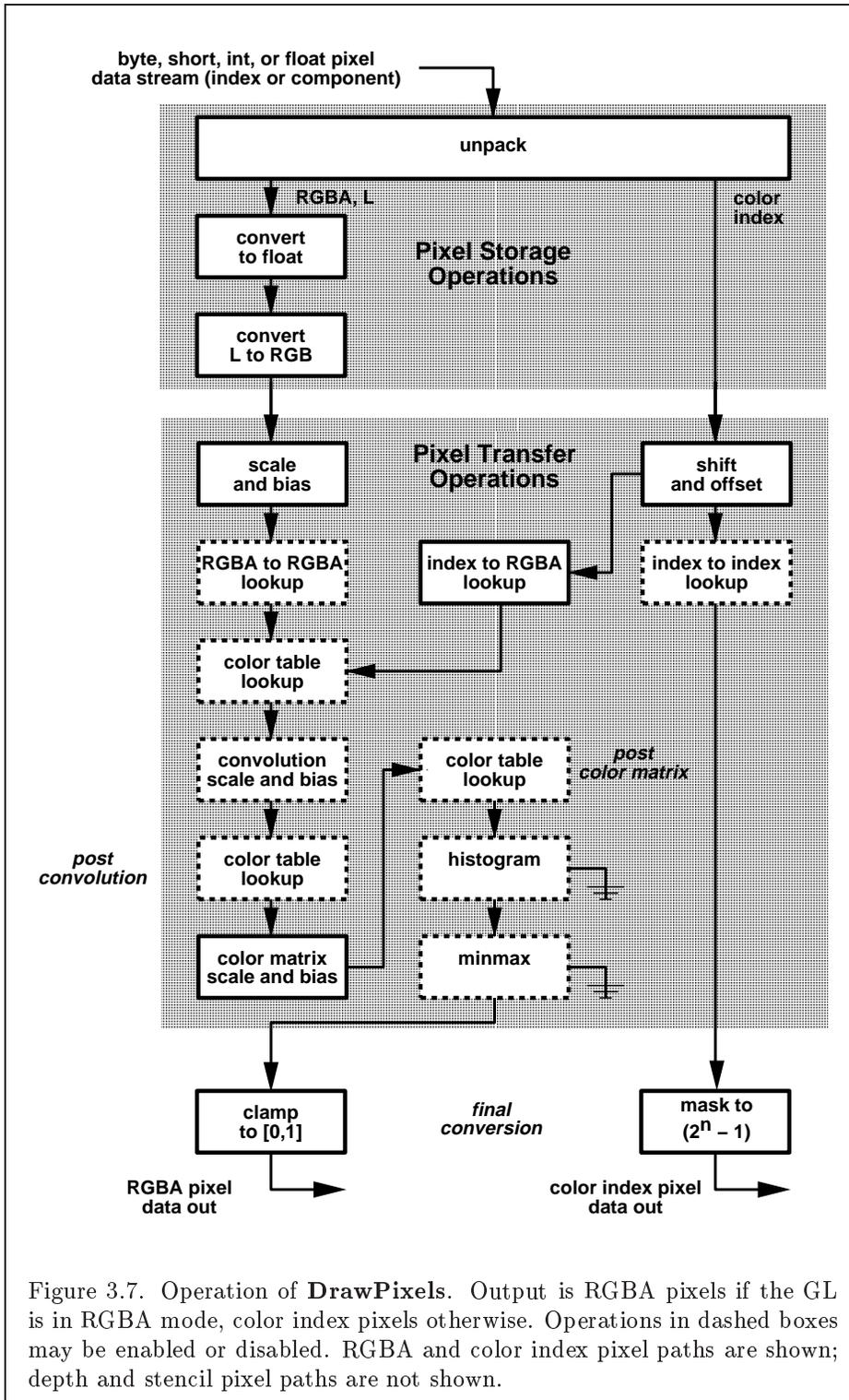


Figure 3.7. Operation of **DrawPixels**. Output is RGBA pixels if the GL is in RGBA mode, color index pixels otherwise. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

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 *N*th row is indicated by

<i>type</i> Parameter Token Name	Corresponding GL Data Type	Special 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_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_10_REV	uint	Yes

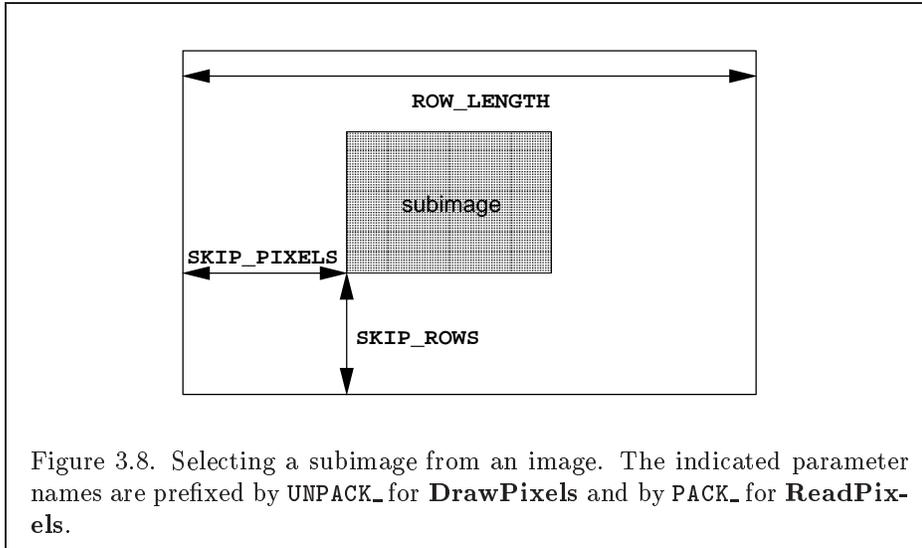
Table 3.5: **DrawPixels** and **ReadPixels** *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6.4.

Format Name	Element Meaning and Order	Target Buffer
COLOR_INDEX	Color Index	Color
STENCIL_INDEX	Stencil Index	Stencil
DEPTH_COMPONENT	Depth	Depth
RED	R	Color
GREEN	G	Color
BLUE	B	Color
ALPHA	A	Color
RGB	R, G, B	Color
RGBA	R, G, B, A	Color
BGR	B, G, R	Color
BGRA	B, G, R, 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	[7..0]	[7..0]
16 bit	[15..0]	[7..0][15..8]
32 bit	[31..0]	[7..0][15..8][23..16][31..24]

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.



$$p + Nk \quad (3.8)$$

where  $N$  is the row number (counting from zero) and  $k$  is defined as

$$k = \begin{cases} nl & s \geq a, \\ a/s \lceil snl/a \rceil & s < a \end{cases} \quad (3.9)$$

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  $(\text{UNPACK\_SKIP\_PIXELS})n + (\text{UNPACK\_SKIP\_ROWS})k$  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,

<i>type</i> Parameter Token Name	GL Data Type	Number of Components	Matching 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_4_REV	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_5_5_5_1	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_1_5_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_10_REV	uint	4	RGBA,BGRA

Table 3.8: Packed pixel formats.

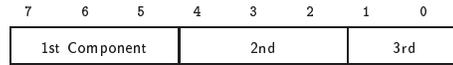
UNSIGNED\_SHORT\_4\_4\_4\_4, UNSIGNED\_SHORT\_4\_4\_4\_4\_REV, UNSIGNED\_SHORT\_5\_5\_5\_1, UNSIGNED\_SHORT\_1\_5\_5\_5\_REV, UNSIGNED\_INT\_8\_8\_8\_8, UNSIGNED\_INT\_8\_8\_8\_8\_REV, UNSIGNED\_INT\_10\_10\_10\_2, or UNSIGNED\_INT\_2\_10\_10\_10\_REV is a special case in which all the components of 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

the most significant bit location of its location in the bitfield.

UNSIGNED\_BYTE\_3\_3\_2:



UNSIGNED\_BYTE\_2\_3\_3\_REV:

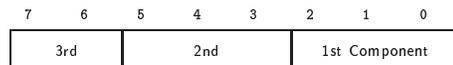
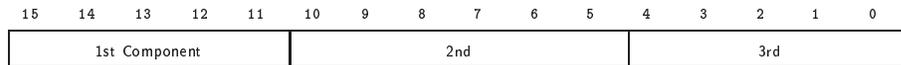
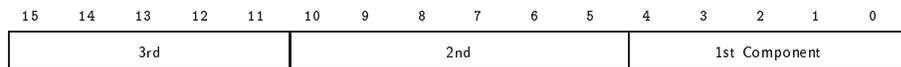


Table 3.9: UNSIGNED\_BYTE formats. Bit numbers are indicated for each component.

UNSIGNED\_SHORT\_5\_6\_5:



UNSIGNED\_SHORT\_5\_6\_5\_REV:



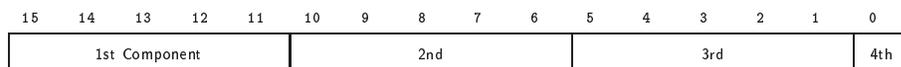
UNSIGNED\_SHORT\_4\_4\_4\_4:



UNSIGNED\_SHORT\_4\_4\_4\_4\_REV:



UNSIGNED\_SHORT\_5\_5\_5\_1:



UNSIGNED\_SHORT\_1\_5\_5\_5\_REV:

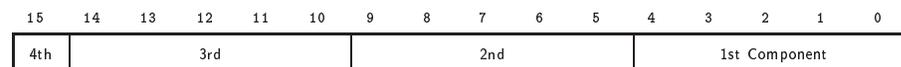


Table 3.10: UNSIGNED\_SHORT formats

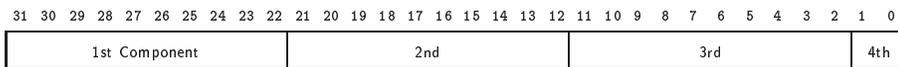
## UNSIGNED\_INT\_8\_8\_8\_8:



## UNSIGNED\_INT\_8\_8\_8\_8\_REV:



## UNSIGNED\_INT\_10\_10\_10\_2:



## UNSIGNED\_INT\_2\_10\_10\_10\_REV:

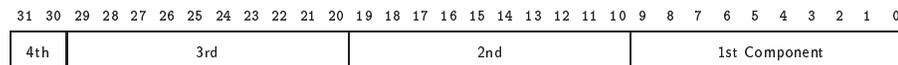


Table 3.11: UNSIGNED\_INT formats

Format	First Component	Second Component	Third Component	Fourth 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

$$k = a \left\lceil \frac{l}{8a} \right\rceil \quad (3.10)$$

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` `ubbytes`. 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  $k$  `ubbytes`. *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

$$(x_{rp} + z_x n, y_{rp} + z_y m) \quad \text{and} \quad (x_{rp} + z_x (n + 1), y_{rp} + z_y (m + 1))$$

(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:

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  $\lceil \log_2(\text{MAX\_PIXEL\_MAP\_TABLE}) \rceil$  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	B	A
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 one-dimensional texture images passed to `TexImage1D`, `TexSubImage1D`, `CopyTexImage1D`, and `CopyTexSubImage1D`, and returned by `GetTexImage` (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`, `CopyTexSubImage2D`, 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	B	A
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 described 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

```
void ConvolutionParameter{if}( enum target,
    enum pname, T param );
```

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  $C'_s[i']$ :

$$C'_s[i'] = \begin{cases} C_s[i'], & 0 \leq i' < W_s \\ C_c, & \textit{otherwise} \end{cases}$$

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  $C'_s[i', j']$ :

$$C'_s[i', j'] = \begin{cases} C_s[i', j'], & 0 \leq i' < W_s, 0 \leq j' < H_s \\ C_c, & \textit{otherwise} \end{cases}$$

**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 one-pixel border of the source image was replicated. Conceptually, each pixel in

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  $C'_s[i']$ :

$$C'_s[i'] = C_s[\text{clamp}(i', W_s)]$$

and the clamping function  $\text{clamp}(val, max)$  is defined as

$$\text{clamp}(val, max) = \begin{cases} 0, & val < 0 \\ val, & 0 \leq val < max \\ max - 1, & val \geq max \end{cases}$$

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  $C'_s[i', j']$ :

$$C'_s[i', j'] = C_s[\text{clamp}(i', W_s), \text{clamp}(j', H_s)]$$

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 component, `POST_CONVOLUTION_BLUE_SCALE` for a B component, and `POST_CONVOLUTION_ALPHA_SCALE` for an A component. The result is added to the corresponding bias: `POST_CONVOLUTION_RED_BIAS`, `POST_CONVOLUTION_GREEN_BIAS`, `POST_CONVOLUTION_BLUE_BIAS`, or `POST_CONVOLUTION_ALPHA_BIAS`.

The required state is three bits indicating whether each of one-dimensional, 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 resulting 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**

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 entries  $B_i$  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*.

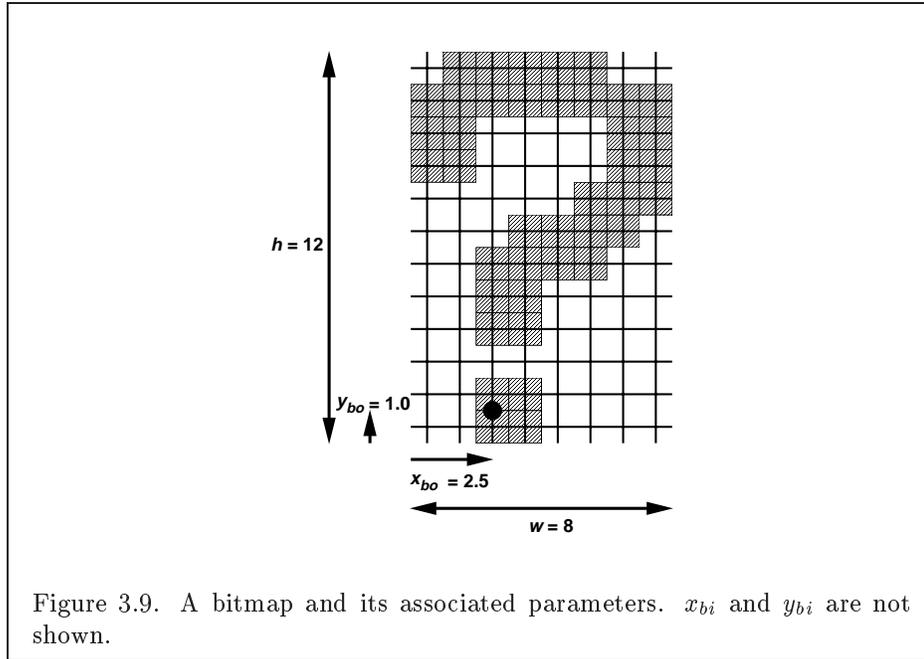
Bitmaps are sent using

```
void Bitmap( sizei w, sizei h, float xbo, float ybo,
             float xbi, float ybi, ubyte *data );
```

*w* and *h* comprise the integer width and height of the rectangular bitmap, respectively. (*x<sub>bo</sub>*, *y<sub>bo</sub>*) gives the floating-point *x* and *y* values of the bitmap's origin. (*x<sub>bi</sub>*, *y<sub>bi</sub>*) 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	A	$A$
LUMINANCE	R	$L$
LUMINANCE_ALPHA	R,A	$L,A$
INTENSITY	R	$I$
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  $N$ th texel is assigned internal integer coordinates  $(i, j, k)$ , where

$$i = (N \bmod \text{width}) - b_s$$

$$j = (\lfloor \frac{N}{\text{width}} \rfloor \bmod \text{height}) - b_s$$

$$k = (\lfloor \frac{N}{\text{width} \times \text{height}} \rfloor \bmod \text{depth}) - b_s$$

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 fixed-point 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  $k \in \{0, 1, \dots, 2^n - 1\}$ , 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

$$w_s = 2^n + 2b_s \tag{3.11}$$

Sized Internal Format	Base Internal Format	<i>R</i> bits	<i>G</i> bits	<i>B</i> bits	<i>A</i> bits	<i>L</i> bits	<i>I</i> 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.

$$h_s = 2^m + 2b_s \quad (3.12)$$

$$d_s = 2^l + 2b_s \quad (3.13)$$

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 two-dimensional texture image, and the maximum allowable height of a two-dimensional 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 three-dimensional image array of level 1 or greater to be created only if a complete<sup>1</sup> 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`.

---

<sup>1</sup>For 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, size_t 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)<sup>2</sup>. 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,

---

<sup>2</sup>Figure 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 two-dimensional 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.

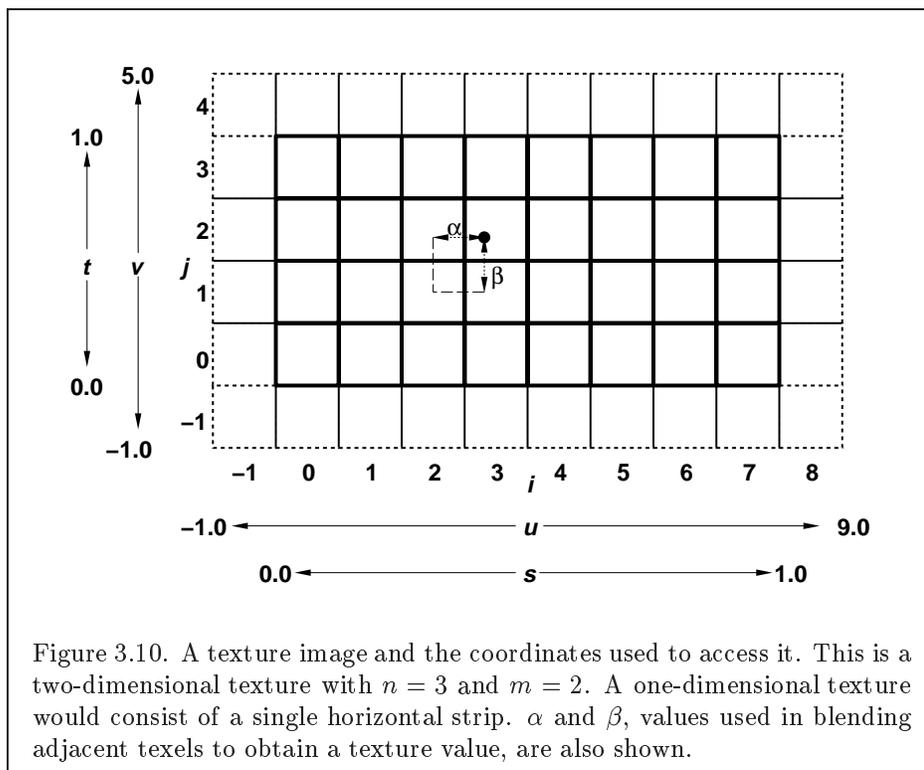
If the *data* argument of **TexImage1D**, **TexImage2D**, or **TexImage3D** is a null pointer (a zero-valued pointer in the C implementation), a one-, two-, or three-dimensional texture array is created with the specified *target*, *level*, *internalformat*, *width*, *height*, and *depth*, but with unspecified 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 );
```



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 **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 **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*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage2D**, except that *internalformat* may not be specified as 1, 2, 3, or 4. An invalid value specified for *internalformat* 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 **TexImage1D**, 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 );
void CopyTexSubImage1D( enum target, int level,
    int xoffset, int x, int y, sizei width );

```

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 **CopyTexSubImage3D** 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**<sup>3</sup>. **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.

---

<sup>3</sup>Because the framebuffer is inherently two-dimensional, there is no **CopyTexImage3D** command.

Arguments *xoffset*, *yoffset*, and *zoffset* of **TexSubImage3D** and **CopyTexSubImage3D** 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**:

$$\begin{aligned} x &< -b_s \\ x + w &> w_s - b_s \\ y &< -b_s \\ y + h &> h_s - b_s \\ z &< -b_s \\ z + d &> d_s - b_s \end{aligned}$$

(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

$$\begin{aligned} i &= x + (n \bmod w) \\ j &= y + (\lfloor \frac{n}{w} \rfloor \bmod h) \\ k &= z + (\lfloor \frac{n}{width * height} \rfloor \bmod d) \end{aligned}$$

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D** 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`:

$$\begin{aligned}x &< -b_s \\x + w &> w_s - b_s \\y &< -b_s \\y + h &> h_s - b_s\end{aligned}$$

(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

$$\begin{aligned}i &= x + (n \bmod w) \\j &= y + (\lfloor \frac{n}{w} \rfloor \bmod h)\end{aligned}$$

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`:

$$\begin{aligned}x &< -b_s \\x + w &> w_s - b_s\end{aligned}$$

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

$$\lambda'(x, y) = \log_2[\rho(x, y)]$$

$$\lambda = \begin{cases} \text{TEXTURE\_MAX\_LOD}, & \lambda' > \text{TEXTURE\_MAX\_LOD} \\ \lambda', & \text{TEXTURE\_MIN\_LOD} \leq \lambda' \leq \text{TEXTURE\_MAX\_LOD} \\ \text{TEXTURE\_MIN\_LOD}, & \lambda' < \text{TEXTURE\_MIN\_LOD} \\ \text{undefined}, & \text{TEXTURE\_MIN\_LOD} > \text{TEXTURE\_MAX\_LOD} \end{cases} \quad (3.14)$$

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  $\lambda$ . 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,  $\rho$  is given at a fragment with window coordinates  $(x, y)$  by

$$\rho = \max \left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right\} \quad (3.15)$$

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

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} / l, \quad (3.16)$$

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  $\rho$  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

$$m_u = \max \left\{ \left| \frac{\partial u}{\partial x} \right|, \left| \frac{\partial u}{\partial y} \right| \right\}$$

$$m_v = \max \left\{ \left| \frac{\partial v}{\partial x} \right|, \left| \frac{\partial v}{\partial y} \right| \right\}$$

$$m_w = \max \left\{ \left| \frac{\partial w}{\partial x} \right|, \left| \frac{\partial w}{\partial y} \right| \right\}.$$

Then  $\max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w$ .

When  $\lambda$  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

$$i = \begin{cases} \lfloor u \rfloor, & s < 1 \\ 2^n - 1, & s = 1 \end{cases} \quad (3.17)$$

(Recall that if `TEXTURE_WRAP_S` is `REPEAT`, then  $0 \leq s < 1$ .) Similarly,  $j$  is found as

$$j = \begin{cases} \lfloor v \rfloor, & t < 1 \\ 2^m - 1, & t = 1 \end{cases} \quad (3.18)$$

and  $k$  is found as

$$k = \begin{cases} \lfloor w \rfloor, & r < 1 \\ 2^l - 1, & r = 1 \end{cases} \quad (3.19)$$

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

$$i_0 = \begin{cases} \lfloor u - 1/2 \rfloor \bmod 2^n, & \text{TEXTURE_WRAP_S is REPEAT} \\ \lfloor u - 1/2 \rfloor, & \text{otherwise} \end{cases}$$

$$j_0 = \begin{cases} [v - 1/2] \bmod 2^m, & \text{TEXTURE\_WRAP\_T is REPEAT} \\ [v - 1/2], & \text{otherwise} \end{cases}$$

and

$$k_0 = \begin{cases} [w - 1/2] \bmod 2^l, & \text{TEXTURE\_WRAP\_R is REPEAT} \\ [w - 1/2], & \text{otherwise} \end{cases}$$

Then

$$i_1 = \begin{cases} (i_0 + 1) \bmod 2^n, & \text{TEXTURE\_WRAP\_S is REPEAT} \\ i_0 + 1, & \text{otherwise} \end{cases}$$

$$j_1 = \begin{cases} (j_0 + 1) \bmod 2^m, & \text{TEXTURE\_WRAP\_T is REPEAT} \\ j_0 + 1, & \text{otherwise} \end{cases}$$

and

$$k_1 = \begin{cases} (k_0 + 1) \bmod 2^l, & \text{TEXTURE\_WRAP\_R is REPEAT} \\ k_0 + 1, & \text{otherwise} \end{cases}$$

Let

$$\alpha = \text{frac}(u - 1/2)$$

$$\beta = \text{frac}(v - 1/2)$$

$$\gamma = \text{frac}(w - 1/2)$$

where  $\text{frac}(x)$  denotes the fractional part of  $x$ .

For a three-dimensional texture, the texture value  $\tau$  is found as

$$\begin{aligned} \tau = & (1 - \alpha)(1 - \beta)(1 - \gamma)\tau_{i_0 j_0 k_0} + \alpha(1 - \beta)(1 - \gamma)\tau_{i_1 j_0 k_0} \\ & + (1 - \alpha)\beta(1 - \gamma)\tau_{i_0 j_1 k_0} + \alpha\beta(1 - \gamma)\tau_{i_1 j_1 k_0} \\ & + (1 - \alpha)(1 - \beta)\gamma\tau_{i_0 j_0 k_1} + \alpha(1 - \beta)\gamma\tau_{i_1 j_0 k_1} \\ & + (1 - \alpha)\beta\gamma\tau_{i_0 j_1 k_1} + \alpha\beta\gamma\tau_{i_1 j_1 k_1} \end{aligned}$$

where  $\tau_{ijk}$  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  $\tau_{ij}$  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  $\tau_i$  is the texel at location  $i$  in the one-dimensional texture.

If any of the selected  $\tau_{ijk}$ ,  $\tau_{ij}$ , or  $\tau_i$  in the above equations refer to a border texel with  $i < -b_s$ ,  $j < -b_s$ ,  $k < -b_s$ ,  $i \geq w_s - b_s$ ,  $j \geq h_s - b_s$ , or  $j \geq d_s - b_s$ , 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

$$\sigma(i - 1) \times \sigma(j - 1) \times \sigma(k - 1)$$

where the dimensions of the previous array are

$$\sigma(i) \times \sigma(j) \times \sigma(k)$$

and

$$\sigma(x) = \begin{cases} 2^x & x > 0 \\ 1 & x \leq 0 \end{cases}$$

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  $p = \max\{n, m, l\} + \text{TEXTURE\_BASE\_LEVEL}$  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  $\text{TEXTURE\_MAX\_LEVEL} < \text{TEXTURE\_BASE\_LEVEL}$ , or if  $\text{TEXTURE\_BASE\_LEVEL} > p$ . Array levels  $k$  where  $k < \text{TEXTURE\_BASE\_LEVEL}$  or  $k > q$  are insignificant.

The values of `TEXTURE\_BASE\_LEVEL` and `TEXTURE\_MAX\_LEVEL` may be re-specified 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  $\lambda$  at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of  $\lambda$  where  $\lambda > c$ ). In the following equations, let

$$b = \text{TEXTURE\_BASE\_LEVEL}$$

For mipmap filters `NEAREST\_MIPMAP\_NEAREST` and `LINEAR\_MIPMAP\_NEAREST`, the  $d$ th mipmap array is selected, where

$$d = \begin{cases} b, & \lambda \leq \frac{1}{2} \\ \lceil b + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, b + \lambda \leq q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, b + \lambda > q + \frac{1}{2} \end{cases} \quad (3.21)$$

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

$$d_1 = \begin{cases} q, & b + \lambda \geq q \\ \lceil b + \lambda \rceil, & \textit{otherwise} \end{cases} \quad (3.22)$$

$$d_2 = \begin{cases} q, & b + \lambda \geq q \\ d_1 + 1, & \textit{otherwise} \end{cases} \quad (3.23)$$

The rules for `NEAREST` or `LINEAR` filtering are then applied to each of the selected arrays, yielding two corresponding texture values  $\tau_1$  and  $\tau_2$ . The final texture value is then found as

$$\tau = [1 - \text{frac}(\lambda)]\tau_1 + \text{frac}(\lambda)\tau_2.$$

### 3.8.6 Texture Magnification

When  $\lambda$  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 switch-over 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 (two- or 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 (two- and 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 **GetTexParameteriv** 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

```
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 three-dimensional 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 **GetTexParameteriv** 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

```
void PrioritizeTextures( sizei n, uint *textures,
                        clampf *priorities );
```

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 Internal Format	REPLACE Texture Function	MODULATE Texture Function
ALPHA	$R_v = R_f$ $G_v = G_f$ $B_v = B_f$ $A_v = A_t$	$R_v = R_f$ $G_v = G_f$ $B_v = B_f$ $A_v = A_f A_t$
LUMINANCE (or 1)	$R_v = L_t$ $G_v = L_t$ $B_v = L_t$ $A_v = A_f$	$R_v = R_f L_t$ $G_v = G_f L_t$ $B_v = B_f L_t$ $A_v = A_f$
LUMINANCE_ALPHA (or 2)	$R_v = L_t$ $G_v = L_t$ $B_v = L_t$ $A_v = A_t$	$R_v = R_f L_t$ $G_v = G_f L_t$ $B_v = B_f L_t$ $A_v = A_f A_t$
INTENSITY	$R_v = I_t$ $G_v = I_t$ $B_v = I_t$ $A_v = I_t$	$R_v = R_f I_t$ $G_v = G_f I_t$ $B_v = B_f I_t$ $A_v = A_f I_t$
RGB (or 3)	$R_v = R_t$ $G_v = G_t$ $B_v = B_t$ $A_v = A_f$	$R_v = R_f R_t$ $G_v = G_f G_t$ $B_v = B_f B_t$ $A_v = A_f$
RGBA (or 4)	$R_v = R_t$ $G_v = G_t$ $B_v = B_t$ $A_v = A_t$	$R_v = R_f R_t$ $G_v = G_f G_t$ $B_v = B_f B_t$ $A_v = A_f A_t$

Table 3.18: Replace and modulate texture functions.

Base Internal Format	DECAL Texture Function	BLEND Texture Function
ALPHA	undefined	$R_v = R_f$ $G_v = G_f$ $B_v = B_f$ $A_v = A_f A_t$
LUMINANCE (or 1)	undefined	$R_v = R_f(1 - L_t) + R_c L_t$ $G_v = G_f(1 - L_t) + G_c L_t$ $B_v = B_f(1 - L_t) + B_c L_t$ $A_v = A_f$
LUMINANCE_ALPHA (or 2)	undefined	$R_v = R_f(1 - L_t) + R_c L_t$ $G_v = G_f(1 - L_t) + G_c L_t$ $B_v = B_f(1 - L_t) + B_c L_t$ $A_v = A_f A_t$
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 (or 3)	$R_v = R_t$ $G_v = G_t$ $B_v = B_t$ $A_v = A_f$	$R_v = R_f(1 - R_t) + R_c R_t$ $G_v = G_f(1 - G_t) + G_c G_t$ $B_v = B_f(1 - B_t) + B_c B_t$ $A_v = A_f$
RGBA (or 4)	$R_v = R_f(1 - A_t) + R_t A_t$ $G_v = G_f(1 - A_t) + G_t A_t$ $B_v = B_f(1 - A_t) + B_t A_t$ $A_v = A_f$	$R_v = R_f(1 - R_t) + R_c R_t$ $G_v = G_f(1 - G_t) + G_c G_t$ $B_v = B_f(1 - B_t) + B_c B_t$ $A_v = A_f A_t$

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 two-dimensional 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:

$$f = \exp(-d \cdot z), \quad (3.24)$$

$$f = \exp(-(d \cdot z)^2), \text{ or} \quad (3.25)$$

$$f = \frac{e - z}{e - s} \quad (3.26)$$

( $z$  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

```
void Fog{if}( enum pname, T param );
void Fog{if}v( enum pname, T params );
```

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

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.

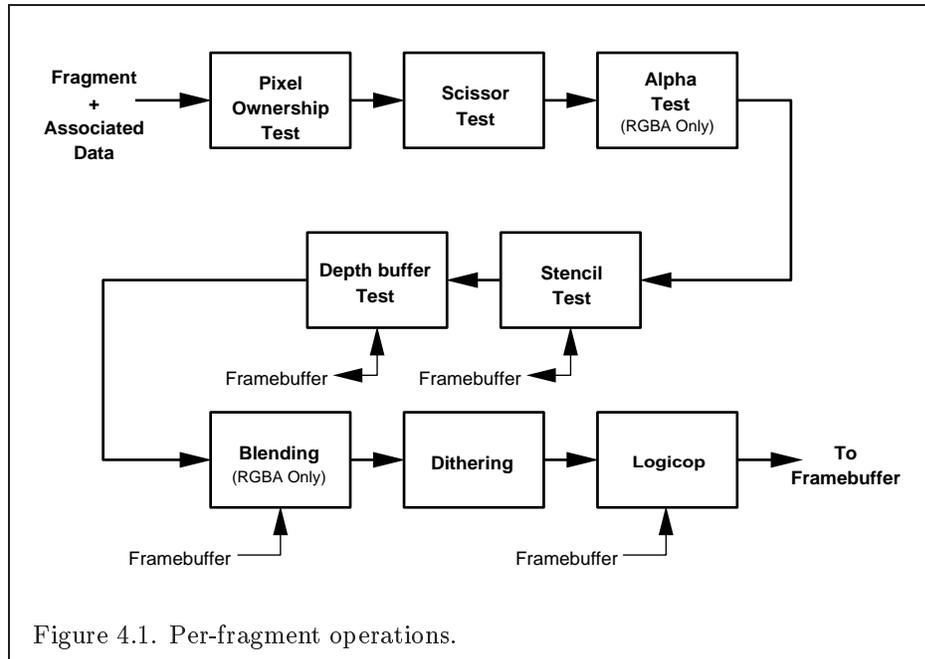
## Chapter 4

# 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

```
void Scissor( int left, int bottom, sizei width,
             sizei height );
```

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 eight-valued 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 dpsfail, 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 (*dpsfail*), 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  $c$  respectively.

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 BlendColor( clampf red, clampf green, clampf blue,
                clampf alpha );
void BlendEquation( enum mode );

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  $x$  and  $y$  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	$s \wedge \neg d$
COPY	$s$
AND_INVERTED	$\neg s \wedge d$
NOOP	$d$
XOR	$s \text{ xor } d$
OR	$s \vee d$
NOR	$\neg(s \vee d)$
EQUIV	$\neg(s \text{ xor } d)$
INVERT	$\neg d$
OR_REVERSE	$s \vee \neg d$
COPY_INVERTED	$\neg s$
OR_INVERTED	$\neg s \vee d$
NAND	$\neg(s \wedge d)$
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 AUX0 through AUX $n$ , 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 AUX $i$  that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments other than AUX $i$  that omit reference to FRONT or BACK refer to both front and back buffers. AUX $i$  enables drawing only to *auxiliary* buffer  $i$ . Each AUX $i$  adheres to  $\text{AUX}i = \text{AUX}0 + i$ . The constants and the buffers they indicate are summarized in Table 4.4. If **DrawBuffer** 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 constant	front left	front right	back left	back right	aux <i>i</i>
NONE					
FRONT_LEFT	•				
FRONT_RIGHT		•			
BACK_LEFT			•		
BACK_RIGHT				•	
FRONT	•	•			
BACK			•	•	
LEFT	•		•		
RIGHT		•		•	
FRONT_AND_BACK	•	•	•	•	
AUX <sub><i>i</i></sub>					•

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  $n$  bits 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

```
void IndexMask( uint mask );
void ColorMask( boolean r, boolean g, boolean b,
                 boolean a );
```

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.

```
void ClearColor( clampf r, clampf g, clampf b,
                clampf a );
```

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 **DrawPixels** 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

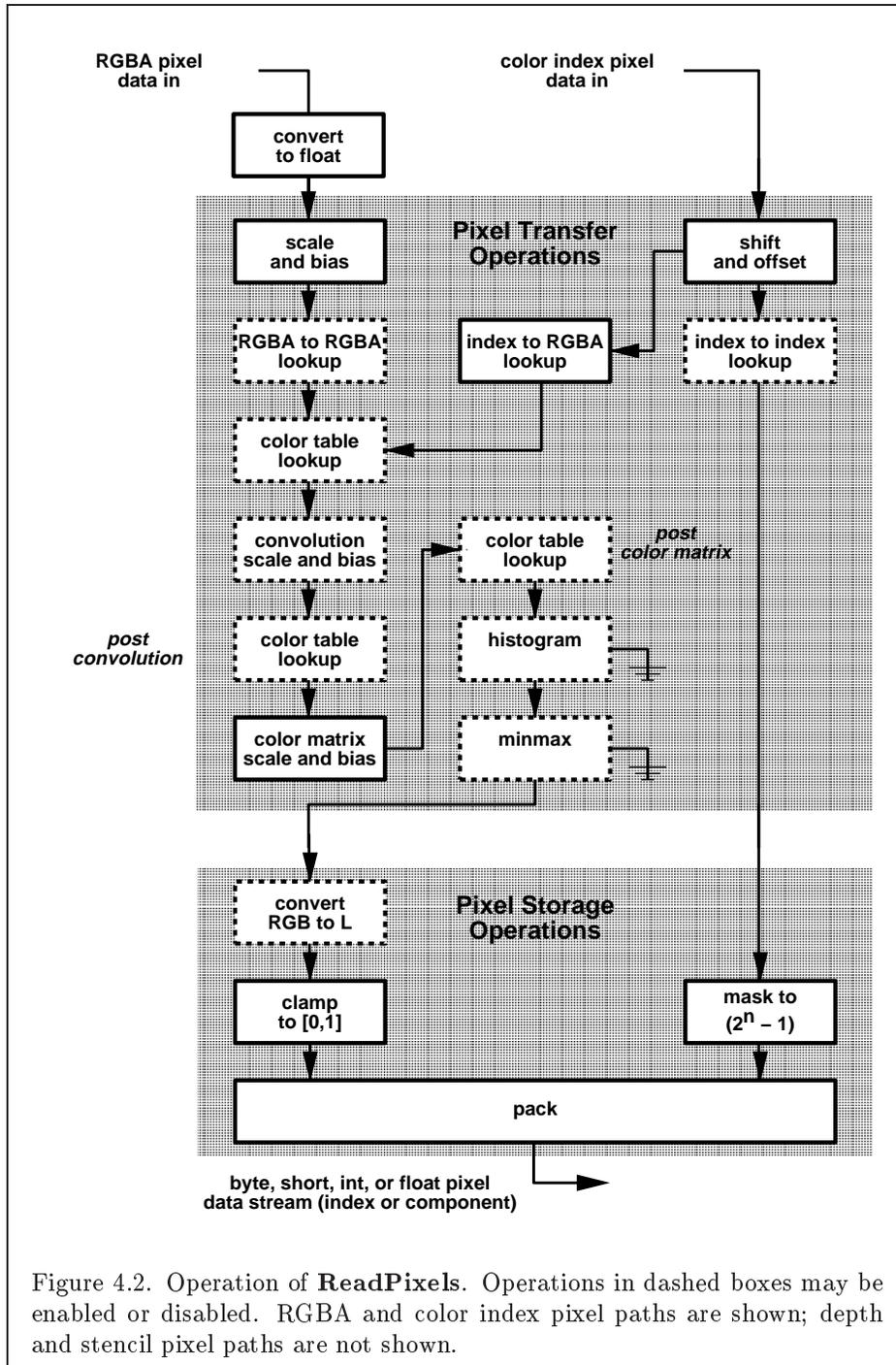
The operation of **DrawPixels** was described in section 3.6.4, except if the *format* argument was **STENCIL\_INDEX**. In this case, all operations described for **DrawPixels** take place, but window  $(x, y)$  coordinates, each with the corresponding stencil index, are produced in lieu of fragments. Each coordinate-stencil 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



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, ∞)
PACK_SKIP_ROWS	integer	0	[0, ∞)
PACK_SKIP_PIXELS	integer	0	[0, ∞)
PACK_ALIGNMENT	integer	4	1,2,4,8
PACK_IMAGE_HEIGHT	integer	0	[0, ∞)
PACK_SKIP_IMAGES	integer	0	[0, ∞)

Table 4.5: **PixelStore** parameters pertaining to **ReadPixels**, **GetTexImage1D**, **GetTexImage2D**, **GetTexImage3D**, **GetColorTable**, **GetConvolutionFilter**, **GetSeparableFilter**, **GetHistogram**, and **GetMinmax**.

```
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 **DrawPixels**. 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 **AUX0** 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  $0 \leq i < width$  and  $0 \leq j < height$ ; 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.

<i>type</i> 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,

<i>type</i> 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_4	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_4_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.

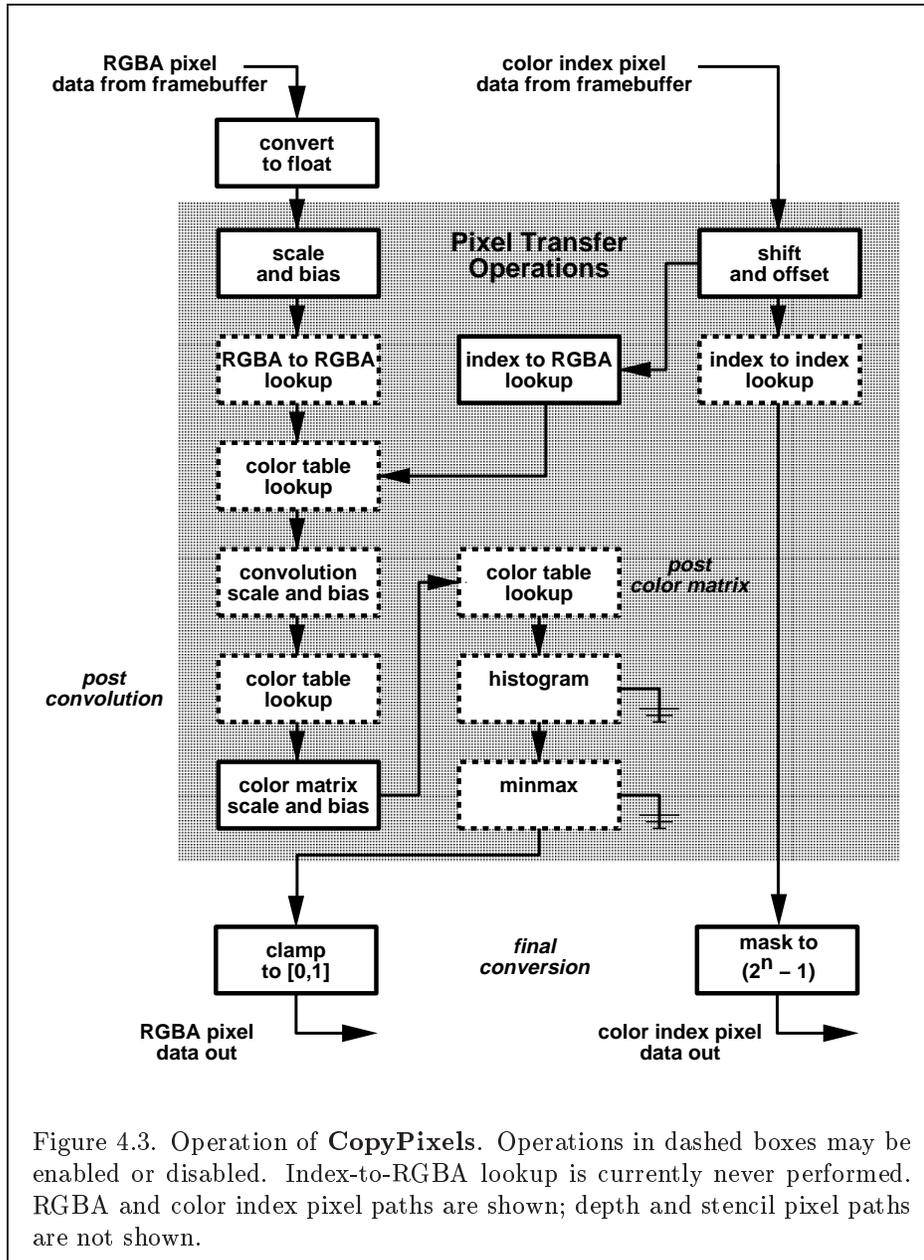


Figure 4.3. Operation of **CopyPixels**. Operations in dashed boxes may be enabled or disabled. Index-to-RGBA lookup is currently never performed. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

## 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

$$\mathbf{p}(u) = \sum_{i=0}^n B_i^n(u) \mathbf{R}_i \tag{5.1}$$

with  $\mathbf{R}_i \in R^k$  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

<i>target</i>	<i>k</i>	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.

```
void Map1{fd}( enum type, T u1, T u2, int stride,
               int order, T points );
```

*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

$$\mathbf{p}'(u') = \mathbf{p}\left(\frac{u' - u_1}{u_2 - u_1}\right).$$

The error `INVALID_VALUE` results if  $u_1 = u_2$ .

**Map2** is analogous to **Map1**, except that it describes bivariate poly-

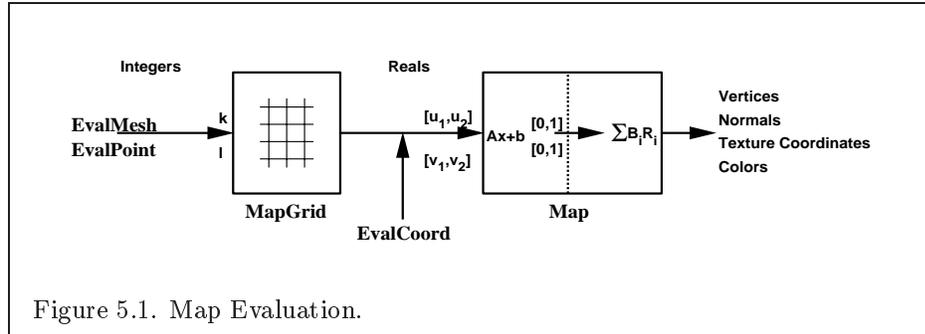


Figure 5.1. Map Evaluation.

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

```
void Map2{fd}( enum target, T u1, T u2, int ustride,
               int vorder, T v1, T v2, int vstride, int vorder, T points );
```

*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

$$\mathbf{p}'(u', v') = \mathbf{p}\left(\frac{u' - u_1}{u_2 - u_1}, \frac{v' - v_1}{v_2 - v_1}\right).$$

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

```
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  $\mathbf{q} = \mathbf{p}$ . For **MAP\_VERTEX\_4**, let  $\mathbf{q} = (x/w, y/w, z/w)$ , where  $(x, y, z, w) = \mathbf{p}$ . Then let

$$\mathbf{m} = \frac{\partial \mathbf{q}}{\partial u} \times \frac{\partial \mathbf{q}}{\partial v}.$$

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 MapGrid1{fd}( int n, T u1', T u2' );
```

for a one-dimensional map or

```
void MapGrid2{fd}( int nu, T u1', T u2', int nv, T v1',
  T v2' );
```

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 p1, int p2 );
```

*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 = p1 to p2 step 1.0
    EvalCoord1( i * Δu' + u1' );
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 p1, int p2, int q1,
  int q2 );
```

*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 * Δu' + u'₁ , q * Δv' + v'₁);
```

The state required for evaluators potentially consists of 9 one-dimensional 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 floating-point 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 model-view 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 *z* coordinates 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

Type	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 commands **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 feedback-item	pixel-rectangle: DRAW_PIXEL_TOKEN vertex COPY_PIXEL_TOKEN vertex
feedback-item:	point line-segment polygon bitmap pixel-rectangle passthrough	passthrough: PASS_THROUGH_TOKEN <i>f</i>
point:	POINT_TOKEN vertex	vertex: 2D: <i>f f</i> 3D: <i>f f f</i> 3D_COLOR: <i>f f f</i> color
line-segment:	LINE_TOKEN vertex vertex LINE_RESET_TOKEN vertex vertex	3D_COLOR_TEXTURE: <i>f f f</i> color tex 4D_COLOR_TEXTURE: <i>f f f f</i> color tex
polygon:	POLYGON_TOKEN <i>n</i> polygon-spec	color: <i>f f f f</i> <i>f</i>
polygon-spec:	polygon-spec vertex vertex vertex vertex	tex: <i>f f f f</i>
bitmap:	BITMAP_TOKEN vertex	

Figure 5.2: Feedback syntax. *f* is a floating-point number. *n* is a floating-point 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( size_t 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 ← 0
for i = 1 to b
    offset ← offset shifted left 8 bits
    offset ← 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  $n, \dots, n + s - 1$  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  $n, \dots, n + s - 1$ , 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**, **IndexPointer**, **TexCoordPointer**, **EdgeFlagPointer**, **InterleavedArrays**, **EnableClientState**, **DisableClientState**, **PushClientAttrib**, **PopClientAttrib**, **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.

### 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 **GetBooleanv** 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 **GetTexGen**, **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 three-dimensional 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 returned. If the *lod* argument is less than zero or if it is larger than 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 three-dimensional 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.

Base Internal Format	R	G	B	A
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 be `CONVOLUTION_1D`, `CONVOLUTION_2D`, or `SEPARABLE_2D`. *pname* is one of `CONVOLUTION_BORDER_COLOR`, `CONVOLUTION_BORDER_MODE`, `CONVOLUTION_FILTER_SCALE`, `CONVOLUTION_FILTER_BIAS`, `CONVOLUTION_FORMAT`, `CONVOLUTION_WIDTH`, `CONVOLUTION_HEIGHT`, `MAX_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 **PopClientAttrib** 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	current	CURRENT_BIT
server	depth-buffer	DEPTH_BUFFER_BIT
server	enable	ENABLE_BIT
server	eval	EVAL_BIT
server	fog	FOG_BIT
server	hint	HINT_BIT
server	lighting	LIGHTING_BIT
server	line	LINE_BIT
server	list	LIST_BIT
server	pixel	PIXEL_MODE_BIT
server	point	POINT_BIT
server	polygon	POLYGON_BIT
server	polygon-stipple	POLYGON_STIPPLE_BIT
server	scissor	SCISSOR_BIT
server	stencil-buffer	STENCIL_BUFFER_BIT
server	texture	TEXTURE_BIT
server	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
$B$	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 $x, y, z$ values)
$V$	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 ( $x, y, z, w$ floating-point coordinates)
$D$	Direction ( $x, y, z$ floating-point coordinates)
$M^4$	$4 \times 4$ 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 minimum)

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 .

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
-	$Z_{11}$	-	0	When $\neq 0$ , indicates <b>begin/end</b> object	2.6.1	-
-	$V$	-	-	Previous vertex in <b>Begin/End line</b>	2.6.1	-
-	$B$	-	-	Indicates if <i>line-vertex</i> is the first	2.6.1	-
-	$V$	-	-	First vertex of a <b>Begin/End line loop</b>	2.6.1	-
-	$Z^+$	-	-	Line stipple counter	3.4	-
-	$n \times V$	-	-	Vertices inside of <b>Begin/End polygon</b>	2.6.1	-
-	$Z^+$	-	-	Number of <i>polygon-vertices</i>	2.6.1	-
-	$2 \times V$	-	-	Previous two vertices in a <b>Begin/End triangle strip</b>	2.6.1	-
-	$Z_3$	-	-	Number of vertices so far in triangle strip: 0, 1, or more	2.6.1	-
-	$Z_2$	-	-	Triangle strip A/B vertex pointer	2.6.1	-
-	$3 \times V$	-	-	Vertices of the quad under construction	2.6.1	-
-	$Z_4$	-	-	Number of vertices so far in quad strip: 0, 1, 2, or more	2.6.1	-

Table 6.4. GL Internal begin-end state variables (inaccessible)

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	<i>C</i>	<b>GetIntegerv, GetFloatv</b>	1,1,1,1	Current color	2.7	current
CURRENT_INDEX	<i>CI</i>	<b>GetIntegerv, GetFloatv</b>	1	Current color index	2.7	current
CURRENT_TEXTURE_COORDS	<i>T</i>	<b>GetFloatv</b>	0,0,0,1	Current texture coordinates	2.7	current
CURRENT_NORMAL	<i>N</i>	<b>GetFloatv</b>	0,0,1	Current normal	2.7	current
-	<i>C</i>	-	-	Color associated with last vertex	2.6	-
-	<i>CI</i>	-	-	Color index associated with last vertex	2.6	-
-	<i>T</i>	-	-	Texture coordinates associated with last vertex	2.6	-
CURRENT_RASTER_POSITION	$R^4$	<b>GetFloatv</b>	0,0,0,1	Current raster position	2.12	current
CURRENT_RASTER_DISTANCE	$R^+$	<b>GetFloatv</b>	0	Current raster distance	2.12	current
CURRENT_RASTER_COLOR	<i>C</i>	<b>GetIntegerv, GetFloatv</b>	1,1,1,1	Color associated with raster position	2.12	current
CURRENT_RASTER_INDEX	<i>CI</i>	<b>GetIntegerv, GetFloatv</b>	1	Color index associated with raster position	2.12	current
CURRENT_RASTER_TEXTURE_COORDS	<i>T</i>	<b>GetFloatv</b>	0,0,0,1	Texture coordinates associated with raster position	2.12	current
CURRENT_RASTER_POSITION_VALID	<i>B</i>	<b>GetBooleanv</b>	<i>True</i>	Raster position valid bit	2.12	current
EDGE_FLAG	<i>B</i>	<b>GetBooleanv</b>	<i>True</i>	Edge flag	2.6.2	current

Table 6.5. Current Values and Associated Data

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	Vertex array enable	2.8	vertex-array
VERTEX_ARRAY_SIZE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	4	Coordinates per vertex	2.8	vertex-array
VERTEX_ARRAY_TYPE	<i>Z</i> <sub>4</sub>	<b>GetInteger</b>	FLOAT	Type of vertex coordinates	2.8	vertex-array
VERTEX_ARRAY_STRIDE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	0	Stride between vertices	2.8	vertex-array
VERTEX_ARRAY_POINTER	<i>Y</i>	<b>GetPointer</b>	0	Pointer to the vertex array	2.8	vertex-array
NORMAL_ARRAY	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	Normal array enable	2.8	vertex-array
NORMAL_ARRAY_TYPE	<i>Z</i> <sub>5</sub>	<b>GetInteger</b>	FLOAT	Type of normal coordinates	2.8	vertex-array
NORMAL_ARRAY_STRIDE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	0	Stride between normals	2.8	vertex-array
NORMAL_ARRAY_POINTER	<i>Y</i>	<b>GetPointer</b>	0	Pointer to the normal array	2.8	vertex-array
COLOR_ARRAY	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	Color array enable	2.8	vertex-array
COLOR_ARRAY_SIZE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	4	Colors per vertex	2.8	vertex-array
COLOR_ARRAY_TYPE	<i>Z</i> <sub>8</sub>	<b>GetInteger</b>	FLOAT	Type of color components	2.8	vertex-array
COLOR_ARRAY_STRIDE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	0	Stride between colors	2.8	vertex-array
COLOR_ARRAY_POINTER	<i>Y</i>	<b>GetPointer</b>	0	Pointer to the color array	2.8	vertex-array
INDEX_ARRAY	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	Index array enable	2.8	vertex-array
INDEX_ARRAY_TYPE	<i>Z</i> <sub>4</sub>	<b>GetInteger</b>	FLOAT	Type of indices	2.8	vertex-array
INDEX_ARRAY_STRIDE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	0	Stride between indices	2.8	vertex-array
INDEX_ARRAY_POINTER	<i>Y</i>	<b>GetPointer</b>	0	Pointer to the index array	2.8	vertex-array
TEXTURE_COORD_ARRAY	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	Texture coordinate array enable	2.8	vertex-array
TEXTURE_COORD_ARRAY_SIZE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	4	Coordinates per element	2.8	vertex-array
TEXTURE_COORD_ARRAY_TYPE	<i>Z</i> <sub>4</sub>	<b>GetInteger</b>	FLOAT	Type of texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_STRIDE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	0	Stride between texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_POINTER	<i>Y</i>	<b>GetPointer</b>	0	Pointer to the texture coordinate array	2.8	vertex-array
EDGE_FLAG_ARRAY	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	Edge flag array enable	2.8	vertex-array
EDGE_FLAG_ARRAY_STRIDE	<i>Z</i> <sup>+</sup>	<b>GetInteger</b>	0	Stride between edge flags	2.8	vertex-array
EDGE_FLAG_ARRAY_POINTER	<i>Y</i>	<b>GetPointer</b>	0	Pointer to the edge flag array	2.8	vertex-array

Table 6.6. Vertex Array Data

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
COLOR_MATRIX	$2 * \times M^4$	<b>GetFloatv</b>	Identity	Color matrix stack	3.6.3	—
MODELVIEW_MATRIX	$32 * \times M^4$	<b>GetFloatv</b>	Identity	Model-view matrix stack	2.10.2	—
PROJECTION_MATRIX	$2 * \times M^4$	<b>GetFloatv</b>	Identity	Projection matrix stack	2.10.2	—
TEXTURE_MATRIX	$2 * \times M^4$	<b>GetFloatv</b>	Identity	Texture matrix stack	2.10.2	—
VIEWPORT	$4 \times Z$	<b>GetIntegerv</b>	see 2.10.1	Viewport origin & extent	2.10.1	viewport
DEPTH_RANGE	$2 \times R^+$	<b>GetFloatv</b>	0,1	Depth range near & far	2.10.1	viewport
COLOR_MATRIX_STACK_DEPTH	$Z^+$	<b>GetIntegerv</b>	1	Color matrix stack pointer	3.6.3	—
MODELVIEW_STACK_DEPTH	$Z^+$	<b>GetIntegerv</b>	1	Model-view matrix stack pointer	2.10.2	—
PROJECTION_STACK_DEPTH	$Z^+$	<b>GetIntegerv</b>	1	Projection matrix stack pointer	2.10.2	—
TEXTURE_STACK_DEPTH	$Z^+$	<b>GetIntegerv</b>	1	Texture matrix stack pointer	2.10.2	—
MATRIX_MODE	$Z_4$	<b>GetIntegerv</b>	MODELVIEW	Current matrix mode	2.10.2	transform
NORMALIZE	$B$	<b>IsEnabled</b>	<i>False</i>	Current normal normalization on/off	2.10.3	transform/enable
RESCALE_NORMAL	$B$	<b>IsEnabled</b>	<i>False</i>	Current normal rescaling on/off	2.10.3	transform/enable
CLIP_PLANE <sub><i>i</i></sub>	$6 * \times R^4$	<b>GetClipPlane</b>	0,0,0,0	User clipping plane coefficients	2.11	transform
CLIP_PLANE <sub><i>i</i></sub>	$6 * \times B$	<b>IsEnabled</b>	<i>False</i>	<i>i</i> th user clipping plane enabled	2.11	transform/enable

Table 6.7. Transformation state

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
FOG-COLOR	<i>C</i>	<b>GetFloatv</b>	0,0,0,0	Fog color	3.10	fog
FOG-INDEX	<i>C/I</i>	<b>GetFloatv</b>	0	Fog index	3.10	fog
FOG-DENSITY	<i>R</i>	<b>GetFloatv</b>	1.0	Exponential fog density	3.10	fog
FOG-START	<i>R</i>	<b>GetFloatv</b>	0.0	Linear fog start	3.10	fog
FOG-END	<i>R</i>	<b>GetFloatv</b>	1.0	Linear fog end	3.10	fog
FOG-MODE	<i>Z</i> <sub>3</sub>	<b>GetIntegerv</b>	EXP	Fog mode	3.10	fog
FOG	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	True if fog enabled	3.10	fog/enable
SHADE_MODEL	<i>Z</i> <sup>+</sup>	<b>GetIntegerv</b>	SMOOTH	ShadeModel setting	2.13.7	lighting

Table 6.8. Coloring

Get value	Type	Get Cmmnd	Initial Value	Description	Sec.	Attribute
LIGHTING	$B$	<b>IsEnabled</b>	<i>False</i>	True if lighting is enabled	2.13.1	lighting/enable
COLOR_MATERIAL	$B$	<b>IsEnabled</b>	<i>False</i>	True if color tracking is enabled	2.13.3	lighting/enable
COLOR_MATERIAL_PARAMETER	$Z_5$	<b>GetIntegerv</b>	AMBIENT_AND_DIFFUSE	Material properties tracking current color	2.13.3	lighting
COLOR_MATERIAL_FACE	$Z_3$	<b>GetIntegerv</b>	FRONT_AND_BACK	Face(s) affected by color tracking	2.13.3	lighting
AMBIENT	$2 \times C$	<b>GetMaterialfv</b>	(0.2,0.2,0.2,1.0)	Ambient material color	2.13.1	lighting
DIFFUSE	$2 \times C$	<b>GetMaterialfv</b>	(0.8,0.8,0.8,1.0)	Diffuse material color	2.13.1	lighting
SPECULAR	$2 \times C$	<b>GetMaterialfv</b>	(0.0,0.0,0.0,1.0)	Specular material color	2.13.1	lighting
EMISSION	$2 \times C$	<b>GetMaterialfv</b>	(0.0,0.0,0.0,1.0)	Emissive mat. color	2.13.1	lighting
SHININESS	$2 \times R$	<b>GetMaterialfv</b>	0.0	Specular exponent of material	2.13.1	lighting
LIGHT_MODEL_AMBIENT	$C$	<b>GetFloatv</b>	(0.2,0.2,0.2,1.0)	Ambient scene color	2.13.1	lighting
LIGHT_MODEL_LOCAL_VIEWER	$B$	<b>GetBooleanv</b>	<i>False</i>	Viewer is local	2.13.1	lighting
LIGHT_MODEL_TWO_SIDE	$B$	<b>GetBooleanv</b>	<i>False</i>	Use two-sided lighting	2.13.1	lighting
LIGHT_MODEL_COLOR_CONTROL	$Z_2$	<b>GetIntegerv</b>	SINGLE_COLOR	Color control	2.13.1	lighting

Table 6.9. Lighting (see also Table 2.7 for defaults)

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
AMBIENT	$8 * \times C'$	<b>GetLightfv</b>	(0.0,0.0,0.0,1.0)	Ambient intensity of light $i$	2.13.1	lighting
DIFFUSE	$8 * \times C'$	<b>GetLightfv</b>	see 2.5	Diffuse intensity of light $i$	2.13.1	lighting
SPECULAR	$8 * \times C'$	<b>GetLightfv</b>	see 2.5	Specular intensity of light $i$	2.13.1	lighting
POSITION	$8 * \times P$	<b>GetLightfv</b>	(0.0,0.0,1.0,0.0)	Position of light $i$	2.13.1	lighting
CONSTANT_ATTENUATION	$8 * \times R^+$	<b>GetLightfv</b>	1.0	Constant atten. factor	2.13.1	lighting
LINEAR_ATTENUATION	$8 * \times R^+$	<b>GetLightfv</b>	0.0	Linear atten. factor	2.13.1	lighting
QUADRATIC_ATTENUATION	$8 * \times R^+$	<b>GetLightfv</b>	0.0	Quadratic atten. factor	2.13.1	lighting
SPOT_DIRECTION	$8 * \times D$	<b>GetLightfv</b>	(0.0,0.0,-1.0)	Spotlight direction of light $i$	2.13.1	lighting
SPOT_EXPONENT	$8 * \times R^+$	<b>GetLightfv</b>	0.0	Spotlight exponent of light $i$	2.13.1	lighting
SPOT_CUTOFF	$8 * \times R^+$	<b>GetLightfv</b>	180.0	Spot. angle of light $i$	2.13.1	lighting
LIGHT $_i$	$8 * \times B$	<b>IsEnabled</b>	<i>False</i>	True if light $i$ enabled	2.13.1	lighting/enable
COLOR_INDEXES	$2 \times 3 \times R$	<b>GetMaterialfv</b>	0,1,1	$a_m, d_m,$ and $s_m$ for color index lighting	2.13.1	lighting

Table 6.10. Lighting (cont.)

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
POINT_SIZE	$R^+$	GetFloatv	1.0	Point size	3.3	point
POINT_SMOOTH	$B$	IsEnabled	False	Point antialiasing on	3.3	point/enable
LINE_WIDTH	$R^+$	GetFloatv	1.0	Line width	3.4	line
LINE_SMOOTH	$B$	IsEnabled	False	Line antialiasing on	3.4	line/enable
LINE_STIPPLE_PATTERN	$Z^+$	GetIntegerv	1's	Line stipple	3.4.2	line
LINE_STIPPLE_REPEAT	$Z^+$	GetIntegerv	1	Line stipple repeat	3.4.2	line
LINE_STIPPLE	$B$	IsEnabled	False	Line stipple enable	3.4.2	line/enable
CULL_FACE	$B$	IsEnabled	False	Polygon culling enabled	3.5.1	polygon/enable
CULL_FACE_MODE	$Z_3$	GetIntegerv	BACK	Cull front/back facing polygons	3.5.1	polygon
FRONT_FACE	$Z_2$	GetIntegerv	CCW	Polygon frontface CW/CCW indicator	3.5.1	polygon
POLYGON_SMOOTH	$B$	IsEnabled	False	Polygon antialiasing on	3.5	polygon/enable
POLYGON_MODE	$2 \times Z_3$	GetIntegerv	FILL	Polygon rasterization mode (front & back)	3.5.4	polygon
POLYGON_OFFSET_FACTOR	$R$	GetFloatv	0	Polygon offset factor	3.5.5	polygon
POLYGON_OFFSET_UNITS	$R$	GetFloatv	0	Polygon offset bias	3.5.5	polygon
POLYGON_OFFSET_POINT	$B$	IsEnabled	False	Polygon offset enable for POINT mode rasterization	3.5.5	polygon/enable
POLYGON_OFFSET_LINE	$B$	IsEnabled	False	Polygon offset enable for LINE mode rasterization	3.5.5	polygon/enable
POLYGON_OFFSET_FILL	$B$	IsEnabled	False	Polygon offset enable for FILL mode rasterization	3.5.5	polygon/enable
-	$I$	GetPolygonStipple	1's	Polygon stipple	3.5	polygon-stipple
POLYGON_STIPPLE	$B$	IsEnabled	False	Polygon stipple enable	3.5.2	polygon/enable

Table 6.11. Rasterization

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
TEXTURE_ <i>x</i> D	$3 \times B$	IsEnabled	<i>False</i>	True if <i>x</i> D texturing is enabled; <i>x</i> is 1, 2, or 3	3.8.10	texture/enable
TEXTURE_BINDING_ <i>x</i> D	$3 \times Z^+$	GetIntegerv	0	Texture object bound to TEXTURE_ <i>x</i> D	3.8.8	texture
TEXTURE_ <i>x</i> D	$n \times I$	GetTexImage	see 3.8	<i>x</i> D texture image at l.o.d. <i>i</i>	3.8	—
TEXTURE_WIDTH	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's specified width	3.8	—
TEXTURE_HEIGHT	$n \times Z^+$	GetTexLevelParameter	0	2D texture image <i>i</i> 's specified height	3.8	—
TEXTURE_DEPTH	$n \times Z^+$	GetTexLevelParameter	0	3D texture image <i>i</i> 's specified depth	3.8	—
TEXTURE_BORDER	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's specified border width	3.8	—
TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS)	$n \times Z_{42}$	GetTexLevelParameter	1	<i>x</i> D texture image <i>i</i> 's internal image format	3.8	—
TEXTURE_RED_SIZE	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's red resolution	3.8	—
TEXTURE_GREEN_SIZE	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's green resolution	3.8	—
TEXTURE_BLUE_SIZE	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's blue resolution	3.8	—
TEXTURE_ALPHA_SIZE	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's alpha resolution	3.8	—
TEXTURE_LUMINANCE_SIZE	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's luminance resolution	3.8	—
TEXTURE_INTENSITY_SIZE	$n \times Z^+$	GetTexLevelParameter	0	<i>x</i> D texture image <i>i</i> 's intensity resolution	3.8	—

Table 6.12. Texture Objects

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
TEXTURE_BORDER_COLOR	$2^+ \times C$	<b>GetTexParameter</b>	0,0,0,0	Texture border color	3.8	texture
TEXTURE_MIN_FILTER	$2^+ \times Z_6$	<b>GetTexParameter</b>	see 3.8	Texture minification function	3.8.5	texture
TEXTURE_MAG_FILTER	$2^+ \times Z_2$	<b>GetTexParameter</b>	see 3.8	Texture magnification function	3.8.6	texture
TEXTURE_WRAP_S	$3^+ \times Z_3$	<b>GetTexParameter</b>	REPEAT	Texture wrap mode S	3.8	texture
TEXTURE_WRAP_T	$2^+ \times Z_3$	<b>GetTexParameter</b>	REPEAT	Texture wrap mode T	3.8	texture
TEXTURE_WRAP_R	$1^+ \times Z_3$	<b>GetTexParameter</b>	REPEAT	Texture wrap mode R	3.8	texture
TEXTURE_PRIORITY	$2^+ \times R^{[0,1]}$	<b>GetTexParameterfv</b>	1	Texture object priority	3.8.8	texture
TEXTURE_RESIDENT	$2^+ \times B$	<b>GetTexParameteriv</b>	see 3.8.8	Texture residency	3.8.8	texture
TEXTURE_MIN_LOD	$n \times R$	<b>GetTexParameterfv</b>	-1000	Minimum level of detail	3.8	texture
TEXTURE_MAX_LOD	$n \times R$	<b>GetTexParameterfv</b>	1000	Maximum level of detail	3.8	texture
TEXTURE_BASE_LEVEL	$n \times R$	<b>GetTexParameterfv</b>	0	Base texture array	3.8	texture
TEXTURE_MAX_LEVEL	$n \times R$	<b>GetTexParameterfv</b>	1000	Maximum texture array level	3.8	texture

Table 6.13. Texture Objects (cont.)

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
TEXTURE_ENV_MODE	$Z_4$	<b>GetTexEnviv</b>	MODULATE	Texture application function	3.8.9	texture
TEXTURE_ENV_COLOR	$C$	<b>GetTexEnvfv</b>	0,0,0,0	Texture environment color	3.8.9	texture
TEXTURE_GEN_*	$4 \times B$	<b>IsEnabled</b>	<i>False</i>	Texgen enabled ( $x$ is S, T, R, or Q)	2.10.4	texture/enable
EYE_PLANE	$4 \times R^4$	<b>GetTexGenfv</b>	see 2.10.4	Texgen plane equation coefficients (for S, T, R, and Q)	2.10.4	texture
OBJECT_PLANE	$4 \times R^4$	<b>GetTexGenfv</b>	see 2.10.4	Texgen object linear coefficients (for S, T, R, and Q)	2.10.4	texture
TEXTURE_GEN_MODE	$4 \times Z_3$	<b>GetTexGeniv</b>	EYE_LINEAR	Function used for texgen (for S, T, R, and Q)	2.10.4	texture

Table 6.14. Texture Environment and Generation

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
SCISSOR_TEST	$B$	<b>IsEnabled</b>	<i>False</i>	Scissoring enabled	4.1.2	scissor/enable
SCISSOR_BOX	$4 \times Z$	<b>GetIntegerv</b>	see 4.1.2	Scissor box	4.1.2	scissor
ALPHA_TEST	$B$	<b>IsEnabled</b>	<i>False</i>	Alpha test enabled	4.1.3	color-buffer/enable
ALPHA_TEST_FUNC	$Z_8$	<b>GetIntegerv</b>	ALWAYS	Alpha test function	4.1.3	color-buffer
ALPHA_TEST_REF	$R^+$	<b>GetIntegerv</b>	0	Alpha test reference value	4.1.3	color-buffer
STENCIL_TEST	$B$	<b>IsEnabled</b>	<i>False</i>	Stenciling enabled	4.1.4	stencil-buffer/enable
STENCIL_FUNC	$Z_8$	<b>GetIntegerv</b>	ALWAYS	Stencil function	4.1.4	stencil-buffer
STENCIL_VALUE_MASK	$Z^+$	<b>GetIntegerv</b>	1's	Stencil mask	4.1.4	stencil-buffer
STENCIL_REF	$Z^+$	<b>GetIntegerv</b>	0	Stencil reference value	4.1.4	stencil-buffer
STENCIL_FAIL	$Z_6$	<b>GetIntegerv</b>	KEEP	Stencil fail action	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	$Z_6$	<b>GetIntegerv</b>	KEEP	Stencil depth buffer fail action	4.1.4	stencil-buffer
STENCIL_PASS_DEPTH_PASS	$Z_6$	<b>GetIntegerv</b>	KEEP	Stencil depth buffer pass action	4.1.4	stencil-buffer
DEPTH_TEST	$B$	<b>IsEnabled</b>	<i>False</i>	Depth buffer enabled	4.1.5	depth-buffer/enable
DEPTH_FUNC	$Z_8$	<b>GetIntegerv</b>	LESS	Depth buffer test function	4.1.5	depth-buffer
BLEND	$B$	<b>IsEnabled</b>	<i>False</i>	Blending enabled	4.1.6	color-buffer/enable
BLEND_SRC	$Z_{13}$	<b>GetIntegerv</b>	ONE	Blending source function	4.1.6	color-buffer
BLEND_DST	$Z_{12}$	<b>GetIntegerv</b>	ZERO	Blending destination function	4.1.6	color-buffer
BLEND_EQUATION	$Z_5$	<b>GetIntegerv</b>	FUNC_ADD	Blending equation	4.1.6	color-buffer
BLEND_COLOR	$C$	<b>GetFloatv</b>	0,0,0,0	Constant blend color	4.1.6	color-buffer
DITHER	$B$	<b>IsEnabled</b>	<i>True</i>	Dithering enabled	4.1.7	color-buffer/enable
INDEX_LOGIC_OP (v1.0: GL_LOGIC_OP)	$B$	<b>IsEnabled</b>	<i>False</i>	Index logic op enabled	4.1.8	color-buffer/enable
COLOR_LOGIC_OP	$B$	<b>IsEnabled</b>	<i>False</i>	Color logic op enabled	4.1.8	color-buffer/enable
LOGIC_OP_MODE	$Z_{16}$	<b>GetIntegerv</b>	COPY	Logic op function	4.1.8	color-buffer

Table 6.15. Pixel Operations

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
DRAW_BUFFER	$Z_{10}^*$	<b>GetInteger</b>	see 4.2.1	Buffers selected for drawing	4.2.1	color-buffer
INDEX_WRITEMASK	$Z^+$	<b>GetInteger</b>	1's	Color index writemask	4.2.2	color-buffer
COLOR_WRITEMASK	$4 \times B$	<b>GetBoolean</b>	<i>True</i>	Color write enables; R, G, B, or A	4.2.2	color-buffer
DEPTH_WRITEMASK	$B$	<b>GetBoolean</b>	<i>True</i>	Depth buffer enabled for writing	4.2.2	depth-buffer
STENCIL_WRITEMASK	$Z^+$	<b>GetInteger</b>	1's	Stencil buffer writemask	4.2.2	stencil-buffer
COLOR_CLEAR_VALUE	$C$	<b>GetFloatv</b>	0,0,0,0	Color buffer clear value (RGBA mode)	4.2.3	color-buffer
INDEX_CLEAR_VALUE	$CI$	<b>GetFloatv</b>	0	Color buffer clear value (color index mode)	4.2.3	color-buffer
DEPTH_CLEAR_VALUE	$R^+$	<b>GetInteger</b>	1	Depth buffer clear value	4.2.3	depth-buffer
STENCIL_CLEAR_VALUE	$Z^+$	<b>GetInteger</b>	0	Stencil clear value	4.2.3	stencil-buffer
ACCUM_CLEAR_VALUE	$4 \times R^+$	<b>GetFloatv</b>	0	Accumulation buffer clear value	4.2.3	accum-buffer

Table 6.16. Framebuffer Control

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
UNPACK_SWAP_BYTES	<i>B</i>	<b>GetBooleanv</b>	<i>False</i>	Value of UNPACK_SWAP_BYTES	4.3	pixel-store
UNPACK_LSB_FIRST	<i>B</i>	<b>GetBooleanv</b>	<i>False</i>	Value of UNPACK_LSB_FIRST	4.3	pixel-store
UNPACK_IMAGE_HEIGHT	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of UNPACK_IMAGE_HEIGHT	4.3	pixel-store
UNPACK_SKIP_IMAGES	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of UNPACK_SKIP_IMAGES	4.3	pixel-store
UNPACK_ROW_LENGTH	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of UNPACK_ROW_LENGTH	4.3	pixel-store
UNPACK_SKIP_ROWS	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of UNPACK_SKIP_ROWS	4.3	pixel-store
UNPACK_SKIP_PIXELS	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of UNPACK_SKIP_PIXELS	4.3	pixel-store
UNPACK_ALIGNMENT	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	4	Value of UNPACK_ALIGNMENT	4.3	pixel-store
PACK_SWAP_BYTES	<i>B</i>	<b>GetBooleanv</b>	<i>False</i>	Value of PACK_SWAP_BYTES	4.3	pixel-store
PACK_LSB_FIRST	<i>B</i>	<b>GetBooleanv</b>	<i>False</i>	Value of PACK_LSB_FIRST	4.3	pixel-store
PACK_IMAGE_HEIGHT	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of PACK_IMAGE_HEIGHT	4.3	pixel-store
PACK_SKIP_IMAGES	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of PACK_SKIP_IMAGES	4.3	pixel-store
PACK_ROW_LENGTH	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of PACK_ROW_LENGTH	4.3	pixel-store
PACK_SKIP_ROWS	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of PACK_SKIP_ROWS	4.3	pixel-store
PACK_SKIP_PIXELS	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	0	Value of PACK_SKIP_PIXELS	4.3	pixel-store
PACK_ALIGNMENT	<i>Z</i> <sup>+</sup>	<b>GetInterv</b>	4	Value of PACK_ALIGNMENT	4.3	pixel-store

Table 6.17. Pixels

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
MAP_COLOR	$B$	GetBooleanv	<i>False</i>	True if colors are mapped	4.3	pixel
MAP_STENCIL	$B$	GetBooleanv	<i>False</i>	True if stencil values are mapped	4.3	pixel
INDEX_SHIFT	$Z$	GetIntegerv	0	Value of INDEX_SHIFT	4.3	pixel
INDEX_OFFSET	$Z$	GetIntegerv	0	Value of INDEX_OFFSET	4.3	pixel
$x$ _SCALE	$R$	GetFloatv	1	Value of $x$ _SCALE; $x$ is RED, GREEN, BLUE, ALPHA, or DEPTH	4.3	pixel
$x$ _BIAS	$R$	GetFloatv	0	Value of $x$ _BIAS; $x$ is one of RED, GREEN, BLUE, ALPHA, or DEPTH	4.3	pixel
COLOR_TABLE	$B$	IsEnabled	<i>False</i>	True if color table lookup is done	3.6.3	pixel/enable
POST_CONVOLUTION_COLOR_TABLE	$B$	IsEnabled	<i>False</i>	True if post convolution color table lookup is done	3.6.3	pixel/enable
POST_COLOR_MATRIX_COLOR_TABLE	$B$	IsEnabled	<i>False</i>	True if post color matrix color table lookup is done	3.6.3	pixel/enable
COLOR_TABLE	$3 \times I$	GetColorTable	<i>empty</i>	Color tables	3.6.3	–
COLOR_TABLE_FORMAT	$2 \times 3 \times Z_{42}$	GetColorTableParameteriv	RGBA	Color tables' internal image format	3.6.3	–
COLOR_TABLE_WIDTH	$2 \times 3 \times Z^+$	GetColorTableParameteriv	0	Color tables' specified width	3.6.3	–
COLOR_TABLE_ $x$ _SIZE	$6 \times 2 \times 3 \times Z^+$	GetColorTableParameteriv	0	Color table component resolution; $x$ is RED, GREEN, BLUE, ALPHA, LUMINANCE, or INTENSITY	3.6.3	–
COLOR_TABLE_SCALE	$3 \times R^4$	GetColorTableParameterfv	1,1,1,1	Scale factors applied to color table entries	3.6.3	pixel
COLOR_TABLE_BIAS	$3 \times R^4$	GetColorTableParameterfv	0,0,0,0	Bias factors applied to color table entries	3.6.3	pixel

Table 6.18. Pixels (cont.)

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
CONVOLUTION_1D	$B$	<b>IsEnabled</b>	<i>False</i>	True if 1D convolution is done	3.6.3	pixel/enable
CONVOLUTION_2D	$B$	<b>IsEnabled</b>	<i>False</i>	True if 2D convolution is done	3.6.3	pixel/enable
SEPARABLE_2D	$B$	<b>IsEnabled</b>	<i>False</i>	True if separable 2D convolution is done	3.6.3	pixel/enable
CONVOLUTION	$2 \times I$	<b>GetConvolution-Filter</b>	<i>empty</i>	Convolution filters	3.6.3	–
CONVOLUTION	$2 \times I$	<b>GetSeparable-Filter</b>	<i>empty</i>	Separable convolution filter	3.6.3	–
CONVOLUTION_BORDER_COLOR	$3 \times C$	<b>GetConvolution-Parameterfv</b>	0,0,0,0	Convolution border color	4.3	pixel
CONVOLUTION_BORDER_MODE	$3 \times Z_4$	<b>GetConvolution-Parameteriv</b>	REDUCE	Convolution border mode	4.3	pixel
CONVOLUTION_FILTER_SCALE	$3 \times R^4$	<b>GetConvolution-Parameterfv</b>	1,1,1,1	Scale factors applied to convolution filter entries	3.6.3	pixel
CONVOLUTION_FILTER_BIAS	$3 \times R^4$	<b>GetConvolution-Parameterfv</b>	0,0,0,0	Bias factors applied to convolution filter entries	3.6.3	pixel
CONVOLUTION_FORMAT	$3 \times Z_{42}$	<b>GetConvolution-Parameteriv</b>	RGBA	Convolution filter internal format	4.3	–
CONVOLUTION_WIDTH	$3 \times Z^+$	<b>GetConvolution-Parameteriv</b>	0	Convolution filter width	4.3	–
CONVOLUTION_HEIGHT	$2 \times Z^+$	<b>GetConvolution-Parameteriv</b>	0	Convolution filter height	4.3	–

Table 6.19. Pixels (cont.)

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
POST_CONVOLUTION_ <i>x</i> _SCALE	<i>R</i>	<b>GetFloatv</b>	1	Component scale factors after convolution; <i>x</i> is RED, GREEN, BLUE, or ALPHA	3.6.3	pixel
POST_CONVOLUTION_ <i>x</i> _BIAS	<i>R</i>	<b>GetFloatv</b>	0	Component bias factors after convolution; <i>x</i> is RED, GREEN, BLUE, or ALPHA	3.6.3	pixel
POST_COLOR_MATRIX_ <i>x</i> _SCALE	<i>R</i>	<b>GetFloatv</b>	1	Component scale factors after color matrix; <i>x</i> is RED, GREEN, BLUE, or ALPHA	3.6.3	pixel
POST_COLOR_MATRIX_ <i>x</i> _BIAS	<i>R</i>	<b>GetFloatv</b>	0	Component bias factors after color matrix; <i>x</i> is RED, GREEN, BLUE, or ALPHA	3.6.3	pixel
HISTOGRAM	<i>B</i>	<b>IsEnabled</b>	False	True if histogramming is enabled	3.6.3	pixel/enable
HISTOGRAM	<i>I</i>	<b>GetHistogram</b>	<i>empty</i>	Histogram table	3.6.3	–
HISTOGRAM.WIDTH	$2 \times Z^+$	<b>GetHistogram-Parameteriv</b>	0	Histogram table width	3.6.3	–
HISTOGRAM.FORMAT	$2 \times Z_{42}$	<b>GetHistogram-Parameteriv</b>	RGBA	Histogram table internal format	3.6.3	–
HISTOGRAM_ <i>x</i> _SIZE	$5 \times 2 \times Z^+$	<b>GetHistogram-Parameteriv</b>	0	Histogram table component resolution; <i>x</i> is RED, GREEN, BLUE, ALPHA, or LUMINANCE	3.6.3	–
HISTOGRAM_SINK	<i>B</i>	<b>GetHistogram-Parameteriv</b>	False	True if histogramming consumes pixel groups	3.6.3	–

Table 6.20. Pixels (cont.)

Get value	Type	Get Cmmnd	Initial Value	Description	Sec.	Attribute
MINMAX	$B$	IsEnabled	False	True if minmax is enabled	3.6.3	pixel/enable
MINMAX	$R^n$	GetMinMax	$(M,M,M,M),(m,m,m,m)$	Minmax table	3.6.3	–
MINMAX_FORMAT	$Z_{42}$	GetMinMax-Parameteriv	RGBA	Minmax table internal format	3.6.3	–
MINMAX-SINK	$B$	GetMinMax-Parameteriv	False	True if minmax consumes pixel groups	3.6.3	–
ZOOM_X	$R$	GetFloatv	1.0	$x$ zoom factor	4.3	pixel
ZOOM_Y	$R$	GetFloatv	1.0	$y$ zoom factor	4.3	pixel
$x$	$8 \times 32 * \times R$	GetPixelFormat	0's	RGBA <b>PixelFormat</b> translation tables; $x$ is a map name from Table 3.3	4.3	–
$x$	$2 \times 32 * \times Z$	GetPixelFormat	0's	Index <b>PixelFormat</b> translation tables; $x$ is a map name from Table 3.3	4.3	–
$x$ .SIZE	$Z^+$	GetIntegerv	1	Size of table $x$	4.3	–
READ_BUFFER	$Z_3$	GetIntegerv	see 4.3.2	Read source buffer	4.3	pixel

Table 6.21. Pixels (cont.)

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
ORDER	$9 \times Z_{8^*}$	<b>GetMapiv</b>	1	1d map order	5.1	—
ORDER	$9 \times 2 \times Z_{8^*}$	<b>GetMapiv</b>	1,1	2d map orders	5.1	—
COEFF	$9 \times 8^* \times R^n$	<b>GetMapfv</b>	see 5.1	1d control points	5.1	—
COEFF	$9 \times 8^* \times 8^* \times R^n$	<b>GetMapfv</b>	see 5.1	2d control points	5.1	—
DOMAIN	$9 \times 2 \times R$	<b>GetMapfv</b>	see 5.1	1d domain endpoints	5.1	—
DOMAIN	$9 \times 4 \times R$	<b>GetMapfv</b>	see 5.1	2d domain endpoints	5.1	—
MAP1_ <i>x</i>	$9 \times B$	<b>IsEnabled</b>	<i>False</i>	1d map enables: <i>x</i> is map type	5.1	eval/enable
MAP2_ <i>x</i>	$9 \times B$	<b>IsEnabled</b>	<i>False</i>	2d map enables: <i>x</i> is map type	5.1	eval/enable
MAP1_GRID_DOMAIN	$2 \times R$	<b>GetFloatv</b>	0,1	1d grid endpoints	5.1	eval
MAP2_GRID_DOMAIN	$4 \times R$	<b>GetFloatv</b>	0,1;0,1	2d grid endpoints	5.1	eval
MAP1_GRID_SEGMENTS	$Z^+$	<b>GetFloatv</b>	1	1d grid divisions	5.1	eval
MAP2_GRID_SEGMENTS	$2 \times Z^+$	<b>GetFloatv</b>	1,1	2d grid divisions	5.1	eval
AUTO_NORMAL	$B$	<b>IsEnabled</b>	<i>False</i>	True if automatic normal generation enabled	5.1	eval/enable

Table 6.22. Evaluators (**GetMap** takes a map name)

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.	Attribute
PERSPECTIVE-CORRECTION-HINT	Z <sub>3</sub>	GetIntegerv	DONT_CARE	Perspective correction hint	5.6	hint
POINT-SMOOTH-HINT	Z <sub>3</sub>	GetIntegerv	DONT_CARE	Point smooth hint	5.6	hint
LINE-SMOOTH-HINT	Z <sub>3</sub>	GetIntegerv	DONT_CARE	Line smooth hint	5.6	hint
POLYGON-SMOOTH-HINT	Z <sub>3</sub>	GetIntegerv	DONT_CARE	Polygon smooth hint	5.6	hint
FOG-HINT	Z <sub>3</sub>	GetIntegerv	DONT_CARE	Fog hint	5.6	hint

Table 6.23. Hints

Get value	Type	Get Cmd	Minimum Value	Description	Sec.	Attribute
MAX_LIGHTS	$Z^+$	GetIntegerv	8	Maximum number of lights	2.13.1	-
MAX_CLIP_PLANES	$Z^+$	GetIntegerv	6	Maximum number of user clipping planes	2.11	-
MAX_COLOR_MATRIX_STACK_DEPTH	$Z^+$	GetIntegerv	2	Maximum color matrix stack depth	3.6.3	-
MAX_MODELVIEW_STACK_DEPTH	$Z^+$	GetIntegerv	32	Maximum model-view stack depth	2.10.2	-
MAX_PROJECTION_STACK_DEPTH	$Z^+$	GetIntegerv	2	Maximum projection matrix stack depth	2.10.2	-
MAX_TEXTURE_STACK_DEPTH	$Z^+$	GetIntegerv	2	Maximum number depth of texture matrix stack	2.10.2	-
SUBPIXEL_BITS	$Z^+$	GetIntegerv	4	Number of bits of subpixel precision in screen $x_w$ and $y_w$	3	-
MAX_3D_TEXTURE_SIZE	$Z^+$	GetIntegerv	16	See the discussion in Section 3.8.	3.8	-
MAX_TEXTURE_SIZE	$Z^+$	GetIntegerv	64	See the discussion in Section 3.8.	3.8	-
MAX_PIXEL_MAP_TABLE	$Z^+$	GetIntegerv	32	Maximum size of a PixelMap translation table	3.6.3	-
MAX_NAME_STACK_DEPTH	$Z^+$	GetIntegerv	64	Maximum selection name stack depth	5.2	-
MAX_LIST_NESTING	$Z^+$	GetIntegerv	64	Maximum display list call nesting	5.4	-
MAX_EVAL_ORDER	$Z^+$	GetIntegerv	8	Maximum evaluator polynomial order	5.1	-
MAX_VIEWPORT_DIMS	$2 \times Z^+$	GetIntegerv	see 2.10.1	Maximum viewport dimensions	2.10.1	-
MAX_ATTRIB_STACK_DEPTH	$Z^+$	GetIntegerv	16	Maximum depth of the server attribute stack	6	-
MAX_CLIENT_ATTRIB_STACK_DEPTH	$Z^+$	GetIntegerv	16	Maximum depth of the client attribute stack	6	-
-	$3 \times Z^+$	-	32	Maximum size of a color table	3.6.3	-
-	$Z^+$	-	32	Maximum size of the histogram table	3.6.3	-

Table 6.24. Implementation Dependent Values

Get value	Type	Get Cmd	Minimum Value	Description	Sec.	Attribute
AUX_BUFFERS	$Z^+$	GetIntegerv	0	Number of auxiliary buffers	4.2.1	-
RGBA_MODE	$B$	GetBooleanv	-	True if color buffers store rgba	2.7	-
INDEX_MODE	$B$	GetBooleanv	-	True if color buffers store indexes	2.7	-
DOUBLEBUFFER	$B$	GetBooleanv	-	True if front & back buffers exist	4.2.1	-
STEREO	$B$	GetBooleanv	-	True if left & right buffers exist	6	-
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of aliased point sizes	3.3	-
SMOOTH_POINT_SIZE_RANGE (v1.1: POINT_SIZE_RANGE)	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of antialiased point sizes	3.3	-
SMOOTH_POINT_SIZE_GRANULARITY (v1.1: POINT_SIZE_GRANULARITY)	$R^+$	GetFloatv	-	Antialiased point size granularity	3.3	-
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of aliased line widths	3.4	-
SMOOTH_LINE_WIDTH_RANGE (v1.1: LINE_WIDTH_RANGE)	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of antialiased line widths	3.4	-
SMOOTH_LINE_WIDTH_GRANULARITY (v1.1: LINE_WIDTH_GRANULARITY)	$R^+$	GetFloatv	-	Antialiased line width granularity	3.4	-
MAX_CONVOLUTION_WIDTH	$3 \times Z^+$	GetConvolutionParameteriv	3	Maximum width of convolution filter	4.3	-
MAX_CONVOLUTION_HEIGHT	$2 \times Z^+$	GetConvolutionParameteriv	3	Maximum height of convolution filter	4.3	-
MAX_ELEMENTS_INDICES	$Z^+$	GetIntegerv	-	Recommended maximum number of DrawRangeElements indices	2.8	-
MAX_ELEMENTS_VERTICES	$Z^+$	GetIntegerv	-	Recommended maximum number of DrawRangeElements vertices	2.8	-

Table 6.25. More Implementation Dependent Values

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
$x$ -BITS	$Z^+$	<b>GetIntegerv</b>	-	Number of bits in $x$ color buffer component; $x$ is one of RED, GREEN, BLUE, ALPHA, or INDEX	4	-
DEPTHBITS	$Z^+$	<b>GetIntegerv</b>	-	Number of depth buffer planes	4	-
STENCILBITS	$Z^+$	<b>GetIntegerv</b>	-	Number of stencil planes	4	-
ACCUM- $x$ -BITS	$Z^+$	<b>GetIntegerv</b>	-	Number of bits in $x$ accumulation buffer component ( $x$ is RED, GREEN, BLUE, or ALPHA)	4	-

Table 6.26. Implementation Dependent Pixel Depths

Get value	Type	Get Cmd	Initial Value	Description	Sec.	Attribute
LIST_BASE	$Z^+$	GetInteger	0	Setting of <b>ListBase</b>	5.4	list
LIST_INDEX	$Z^+$	GetInteger	0	number of display list under construction; 0 if none	5.4	-
LIST_MODE	$Z^+$	GetInteger	0	Mode of display list under construction; undefined if none	5.4	-
-	$16 * \times A$	-	empty	Server attribute stack	6	-
ATTRIB_STACK_DEPTH	$Z^+$	GetInteger	0	Server attribute stack pointer	6	-
-	$16 * \times A$	-	empty	Client attribute stack	6	-
CLIENT_ATTRIB_STACK_DEPTH	$Z^+$	GetInteger	0	Client attribute stack pointer	6	-
NAME_STACK_DEPTH	$Z^+$	GetInteger	0	Name stack depth	5.2	-
RENDER_MODE	$Z_3$	GetInteger	RENDER	<b>RenderMode</b> setting	5.2	-
SELECTION_BUFFER_POINTER	$Y$	GetPointer	0	Selection buffer pointer	5.2	select
SELECTION_BUFFER_SIZE	$Z^+$	GetInteger	0	Selection buffer size	5.2	select
FEEDBACK_BUFFER_POINTER	$Y$	GetPointer	0	Feedback buffer pointer	5.3	feedback
FEEDBACK_BUFFER_SIZE	$Z^+$	GetInteger	0	Feedback buffer size	5.3	feedback
FEEDBACK_BUFFER_TYPE	$Z_5$	GetInteger	2D	Feedback type	5.3	feedback
-	$n \times Z_8$	GetError	0	Current error code(s)	2.5	-
-	$n \times B$	-	False	True if there is a corresponding error	2.5	-

Table 6.27. Miscellaneous

# 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, high-performance 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, edgeflag)*
- *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 **ColorMaterial** 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 transformations 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  $\lambda$  are added. One constraint clamps  $\lambda$  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  $\lambda$  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. They 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 three-dimensional 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

$$C' = MC,$$

where

$$C = \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

and  $M$  is the  $4 \times 4$  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 occurrences 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 Bisailon, 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 vendor-specific 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 ***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

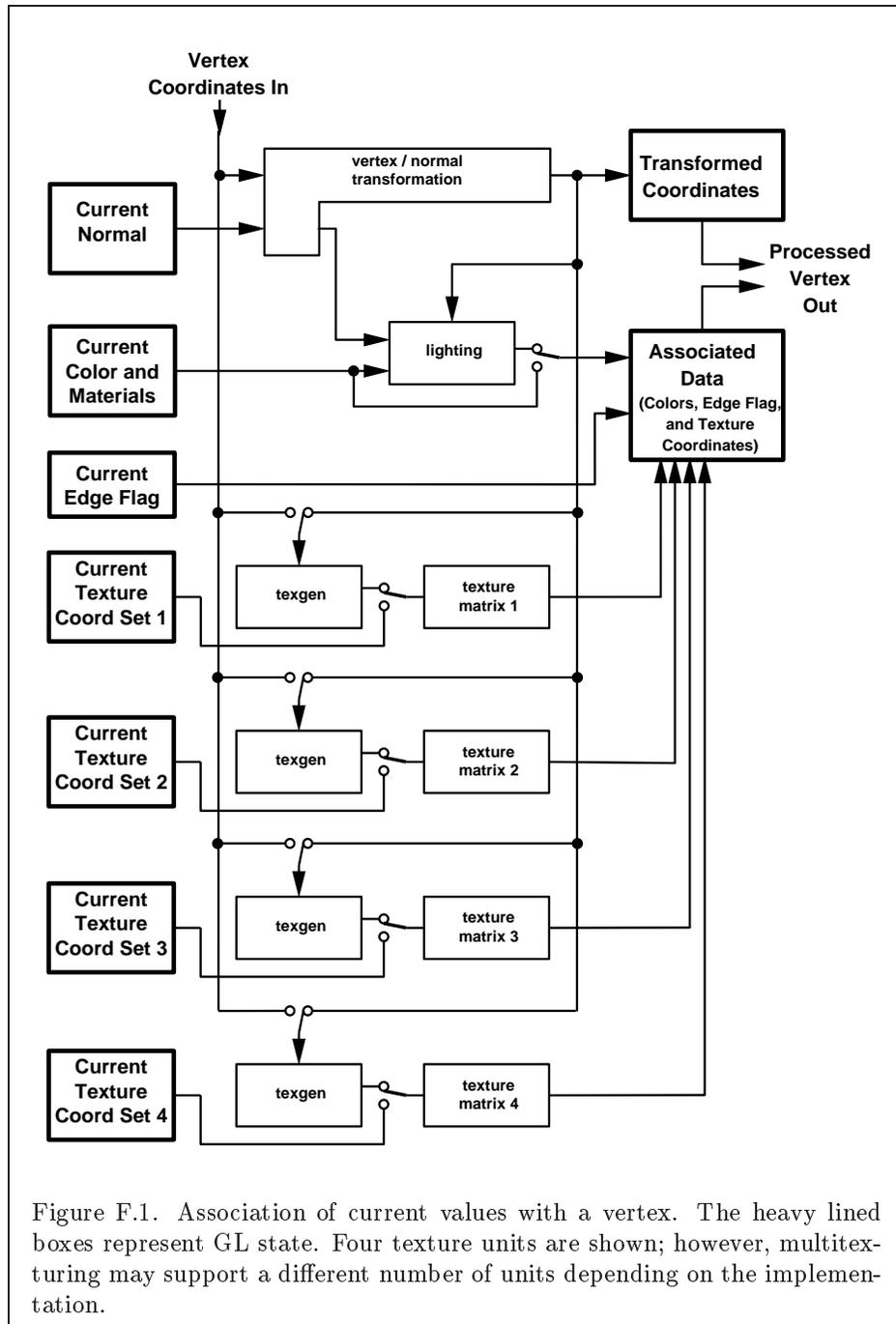
```
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( T coords );
```

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

```
void MultiTexCoord{1234}{sifd}ARB(enum texture, T
    coords)
void MultiTexCoord{1234}{sifd}vARB(enum texture, T
    coords)
```

take the coordinate set to be modified as the *texture* parameter. *texture* is a symbolic constant of the form **TEXTURE<sub>i</sub>\_ARB**, indicating that texture coordinate set *i* is to be modified. The constants obey **TEXTURE<sub>i</sub>\_ARB** =



`TEXTURE0_ARB + i` ( $i$  is in the range 0 to  $k - 1$ , where  $k$  is the implementation-dependent 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 `TEXTURE0_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  $4 \times 4$  matrix is applied to the corresponding texture coordinates. This matrix is applied as

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} \begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix},$$

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 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 model-view 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 four-valued integer indicating the current matrix mode, one stack of at least two  $4 \times 4$  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 `TEXTURE0_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  $(s, t, r)$

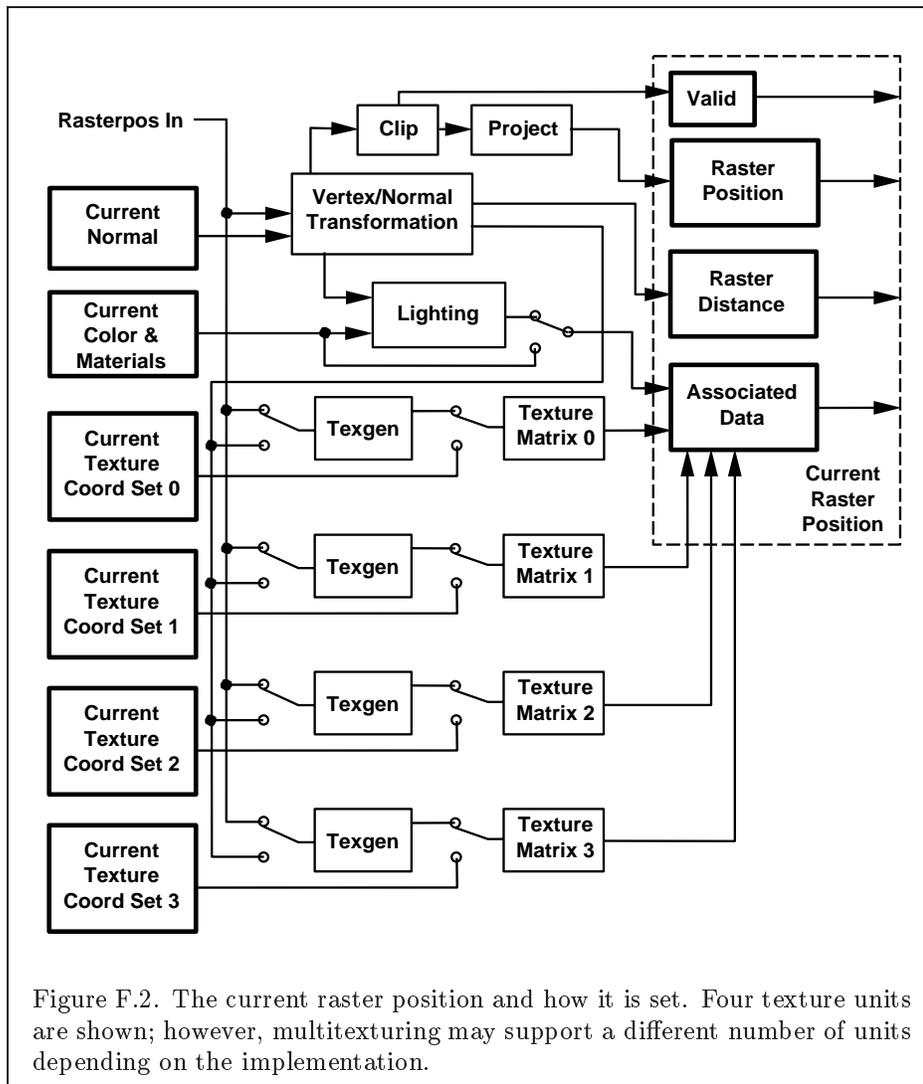


Figure F.2. The current raster position and how it is set. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

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  $p = \max\{n, m, l\} + \text{TEXTURE\_BASE\_LEVEL}$  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 precedence 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.

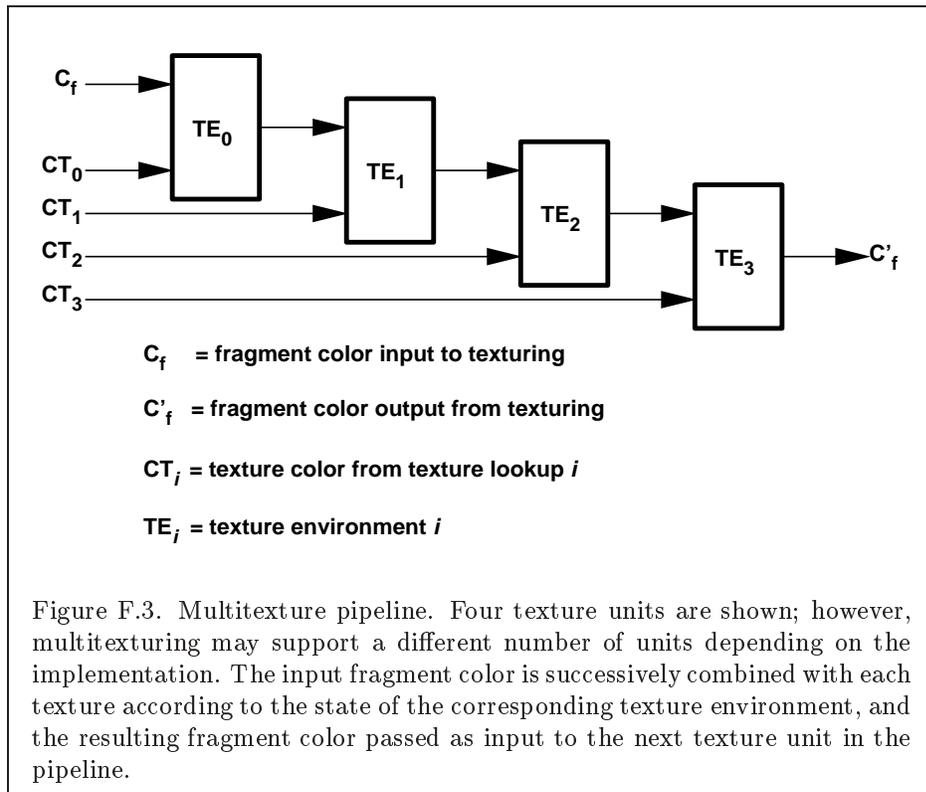
### F.2.13 Changes to Section 5.1 (Evaluators)

*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 **TEXTURE0\_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 **TEXTURE0\_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 `TEXTURE0_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 `TEXTURE0_ARB` is pushed first, followed by state corresponding to `TEXTURE1_ARB`, and so on up to and including the state corresponding to `TEXTURE $k$ _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 $k$ _ARB` and ending with `TEXTURE0_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 Cmnnd	Initial Value	Description	Sec.	Attribute
Modified state in table 6.5						
CURRENT_TEXTURE_COORDS	$1 * \times T$	<b>GetFloatv</b>	0,0,0,1	Current texture coordinates	2.7	current
CURRENT_RASTER_TEXTURE_COORDS	$1 * \times T$	<b>GetFloatv</b>	0,0,0,1	Texture coordinates associated with raster position	2.12	current
Modified state in table 6.6						
TEXTURE_COORD_ARRAY	$1 * \times B$	<b>IsEnabled</b>	<i>False</i>	Texture coordinate array enable	2.8	vertex-array
TEXTURE_COORD_ARRAY_SIZE	$1 * \times Z^+$	<b>GetIntegerv</b>	4	Coordinates per element	2.8	vertex-array
TEXTURE_COORD_ARRAY_TYPE	$1 * \times Z_4$	<b>GetIntegerv</b>	FLOAT	Type of texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_STRIDE	$1 * \times Z^+$	<b>GetIntegerv</b>	0	Stride between texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_POINTER	$1 * \times Y$	<b>GetPointerv</b>	0	Pointer to the texture coordinate array	2.8	vertex-array

Table F.1. Changes to State Tables

Get value	Type	Get Cmdnd	Initial Value	Description	Sec.	Attribute
Modified state in table 6.7						
TEXTURE_MATRIX	$1 * \times 2 * \times M^4$	<b>GetFloatv</b>	Identity	Texture matrix stack	2.10.2	–
TEXTURE_STACK_DEPTH	$1 * \times Z^+$	<b>GetIntegerv</b>	1	Texture matrix stack pointer	2.10.2	–
Modified state in table 6.12						
TEXTURE_xD	$1 * \times 3 \times B$	<b>IsEnabled</b>	<i>False</i>	True if <i>x</i> D texturing is enabled; <i>x</i> is 1, 2, or 3	3.8.10	texture/enable
TEXTURE_BINDING_xD	$1 * \times 3 \times Z^+$	<b>GetIntegerv</b>	0	Texture object bound to TEXTURE_xD	3.8.8	texture
Modified state in table 6.14						
TEXTURE_ENV_MODE	$1 * \times Z_4$	<b>GetTexEnviv</b>	MODULATE	Texture application function	3.8.9	texture
TEXTURE_ENV_COLOR	$1 * \times C$	<b>GetTexEnvfv</b>	0,0,0,0	Texture environment color	3.8.9	texture
TEXTURE_GEN_x	$1 * \times 4 \times B$	<b>IsEnabled</b>	<i>False</i>	Texgen enabled ( <i>x</i> is S, T, R, or Q)	2.10.4	texture/enable
EYE_PLANE	$1 * \times 4 \times R^4$	<b>GetTexGenfv</b>	see 2.10.4	Texgen plane equation coefficients (for S, T, R, and Q)	2.10.4	texture
OBJECT_PLANE	$1 * \times 4 \times R^4$	<b>GetTexGenfv</b>	see 2.10.4	Texgen object linear coefficients (for S, T, R, and Q)	2.10.4	texture
TEXTURE_GEN_MODE	$1 * \times 4 \times Z_3$	<b>GetTexGeniv</b>	EYE_LINEAR	Function used for texgen (for S, T, R, and Q)	2.10.4	texture

Table F.2. Changes to State Tables (cont.)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.	Attribute
Added to table 6.6 CLIENT_ACTIVE_TEXTURE_ARB	$Z_{1*}$	<b>Get Integerv</b>	TEXTURE0_ARB	Client active texture unit selector	2.7	vertex-array
Added to table 6.14 ACTIVE_TEXTURE_ARB	$Z_{1*}$	<b>Get Integerv</b>	TEXTURE0_ARB	Active texture unit selector	2.7	texture

Table F.3. New State Introduced by Multitexture

Get value	Type	Get Cmd	Minimum Value	Description	Sec.	Attribute
Added to table 6.25	$Z^+$	<b>GetIntegerv</b>	1	Number of texture units (not to exceed 32)	2.6	-
MAX_TEXTURE_UNITS_ARB						

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  
  
1, 113, 120, 131, 136, 137, 185, 202, 253  
2, 113, 120, 136, 137, 185, 202, 253  
3, 113, 120, 136, 137, 185, 202, 253  
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, 53  
AND, 151  
AND\_INVERTED, 151  
AND\_REVERSE, 151  
AreTexturesResident, 134, 178  
ArrayElement, 19, 23, 24, 175  
AUTO\_NORMAL, 167  
AUX<sub>*i*</sub>, 151, 152  
AUX<sub>*n*</sub>, 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, 191
- ClientActiveTextureARB, 243
- CLIP\_PLANE<sub>*i*</sub>, 39
- CLIP\_PLANE0, 39
- ClipPlane, 38
- COEFF, 184
- COLOR, 31, 34, 81, 85, 86, 120, 162, 245
- Color, 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, 185
- COLOR\_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, 248
- CopyTexImage2D, 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, 246
- CURRENT\_TEXTURE\_COORDS, 243
- CW, 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, 243
- DITHER, 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, 189
- EdgeFlag, 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, 243
- End, 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, 189
- FeedbackBuffer, 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, 183
- FRONT\_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, 216
- GREEN\_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, 188
- HISTOGRAM\_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
- InterleavedArrays, 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, 249
- INVERT, 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<sub>*i*</sub>, 49, 51, 224
- LIGHT0, 49
- LIGHT\_MODEL\_AMBIENT, 50
- LIGHT\_MODEL\_COLOR\_CONTROL, 50
- LIGHT\_MODEL\_LOCAL\_VIEWER, 50
- LIGHT\_MODEL\_TWO\_SIDE, 50

- 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, 167  
 MAP1\_TEXTURE\_COORD\_2, 165, 167  
 MAP1\_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, 190  
 MAX\_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, 129–131
- NEAREST\_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, 207
- PACK\_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, 178
- PixelTransfer, 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, 78
- POST\_COLOR\_MATRIX\_x\_SCALE, 78
- POST\_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, 78
- 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, 179
- PROXY\_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, 203
- REPLACE, 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, 189
- SEPARABLE\_2D, 85, 103, 117, 187
- SeparableFilter2D, 84
- SEPARATE\_SPECULAR\_COLOR, 47
- SET, 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, 245
- STACK\_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, 184
- STENCIL\_TEST, 144
- StencilFunc, 144, 222
- StencilMask, 153, 156, 223
- StencilOp, 144, 145

- 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, 243
- TexEnv, 135, 249
- TexGen, 36–38, 240, 246
- TexImage, 121
- TexImage1D, 76, 103, 105, 113, 117, 118, 120, 121, 129, 132, 179, 248
- TexImage2D, 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
- TEXTURE<sub>*i*</sub>\_ARB, 241
- TEXTURE0\_ARB, 243, 245, 249, 251, 254
- TEXTURE1\_ARB, 251
- TEXTURE<sub>*x*</sub>D, 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, 183
- TEXTURE\_LUMINANCE\_SIZE, 183
- TEXTURE\_MAG\_FILTER, 124, 131
- TEXTURE\_MAX\_LEVEL, 116, 124, 130, 132, 248
- TEXTURE\_MAX\_LOD, 124–126, 132
- TEXTURE\_MIN\_FILTER, 124, 127, 129–131, 248
- TEXTURE\_MIN\_LOD, 124–126, 132
- TEXTURE\_PRIORITY, 124, 132, 134
- TEXTURE\_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,  
     207  
 UNPACK\_SKIP\_ROWS, 76, 93, 98,  
     207  
 UNPACK\_SWAP\_BYTES, 76, 90, 92,  
     207  
 UNSIGNED\_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,  
     161  
 UNSIGNED\_INT, 22, 24, 91, 97, 160,  
     161, 177  
 UNSIGNED\_INT\_10\_10\_10\_2, 91, 94,  
     97, 161  
 UNSIGNED\_INT\_2\_10\_10\_10\_REV,  
     91, 94, 97, 161  
 UNSIGNED\_INT\_8\_8\_8\_8, 91, 94, 97,  
     161  
 UNSIGNED\_INT\_8\_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\_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<sup>®</sup> 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

<b>1</b>	<b>Overview</b>	<b>1</b>
<b>2</b>	<b>Initialization</b>	<b>2</b>
<b>3</b>	<b>Mipmapping</b>	<b>4</b>
3.1	Image Scaling . . . . .	4
3.2	Automatic Mipmapping . . . . .	5
<b>4</b>	<b>Matrix Manipulation</b>	<b>7</b>
4.1	Matrix Setup . . . . .	7
4.2	Coordinate Projection . . . . .	9
<b>5</b>	<b>Polygon Tessellation</b>	<b>10</b>
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
<b>6</b>	<b>Quadrics</b>	<b>20</b>
6.1	The Quadrics Object . . . . .	20
6.2	Callbacks . . . . .	20
6.3	Rendering Styles . . . . .	21
6.4	Quadrics Primitives . . . . .	22

<b>7</b>	<b>NURBS</b>	<b>24</b>
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
<b>8</b>	<b>Errors</b>	<b>33</b>
<b>9</b>	<b>GLU Versions</b>	<b>34</b>
9.1	GLU 1.1 . . . . .	34
9.2	GLU 1.2 . . . . .	35
9.3	GLU 1.3 . . . . .	35
	<b>Index of GLU Commands</b>	<b>36</b>

# 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

```
GLboolean gluCheckExtension( char *extName,  
                             const GLubyte *extString );
```

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 **glTexImageD**. Other parameters are the same as for **gluBuildDMipmaps**. 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.

`gluOrtho2D` and `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 fovy, GLdouble aspect,  
                    GLdouble near, GLdouble far );
```

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 *viewport* 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

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** / **gluTessEndContour**. 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** / **gluTessEndContour** 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* may 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 errorData( GLenum errno, void *polygon_data );
void combineData( GLdouble coords[3],
                 void *vertex_data[4], GLfloat weight[4], void **outData,
                 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

specified then the callback with *polygon\_data* will be used. Note that *polygon\_data* is a copy of the pointer that was specified when **gluTessBeginPolygon** 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, multi-processor 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:

```
gluBeginPolygon → gluTessBeginPolygon
                  gluTessBeginContour
gluNextContour  → gluTessEndContour
                  gluTessBeginContour
gluEndPolygon   → gluTessEndContour
                  gluTessEndPolygon
```

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**:

```
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 *normals*, *textureCoords*, *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. *textureCoords* 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  $Z = height$ . *baseRadius* specifies the radius at Z

= 0, and *topRadius* specifies the radius at  $Z = \text{height}$ . (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 *outerRadius*, 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 error( GLenum errno );
```

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_STRIP`, `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_AUTO_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. **glPushAttrib** 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 a NURBS surface should be tessellated. *value* may be set to one of `GLU_PATH_LENGTH`, `GLU_PARAMETRIC_ERROR`, `GLU_DOMAIN_DISTANCE`, `GLU_OBJECT_PATH_LENGTH` or `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\_U\_STEP 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:

```
void gluLoadSamplingMatrices( GLUnurbsObj *nurbsObj,
    const GLfloat modelMatrix[16],
    const GLfloat projMatrix[16], const GLint viewport[4] );
```

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 tessellation 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 tessellation 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 tessellation 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 $\alpha$ 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 $\alpha$ 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 tessellations. These features exactly match corresponding features of the `GLU_EXT_nurbs_tessellator` GLU extension, and may be used interchangeably 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

begin, 12, 25  
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\_STRIP, 26  
GL\_LINES, 26  
GL\_MAP\*\_TEXTURE\_COORD\_1, 26  
GL\_MAP\*\_TEXTURE\_COORD\_2, 26  
GL\_MAP\*\_TEXTURE\_COORD\_3, 26  
GL\_MAP\*\_TEXTURE\_COORD\_4, 26  
GL\_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  
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  
glTexImagezD, 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

- 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,  
35
- GLU\_NURBS\_RENDERER,25,30,  
31
- GLU\_NURBS\_TESSELLATOR,25,  
26,30, 32
- GLU\_NURBS\_TEXTURE\_COORD,  
25
- GLU\_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,  
31
- GLU\_OUT\_OF\_MEMORY, 33
- GLU\_OUTLINE\_PATCH,31, 32
- GLU\_OUTLINE\_POLY,31, 32
- GLU\_OUTSIDE,21, 22
- GLU\_PARAMETRIC\_ERROR,30,  
31
- GLU\_PARAMETRIC\_  
TOLERANCE,30,31, 34
- GLU\_PATH\_LENGTH,30,31, 34
- GLU\_POINT, 22
- GLU\_SAMPLING\_METHOD,30,31,  
34, 35
- GLU\_SAMPLING\_TOLERANCE,  
30
- GLU\_SILHOUETTE,22, 23
- GLU\_SMOOTH, 21
- GLU\_TESS\_BEGIN,12,15, 19
- GLU\_TESS\_BEGIN\_DATA,12, 15
- GLU\_TESS\_BOUNDARY\_ONLY,13,  
15, 17
- GLU\_TESS\_COMBINE, 12
- GLU\_TESS\_COMBINE\_DATA, 12
- GLU\_TESS\_COORD\_TOO\_LARGE,  
13
- GLU\_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,  
15
- GLU\_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
- gluBuildzDMipmapLevels, 35
- gluBuildzDMipmaps, 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
- gluPwICurve, 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

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background	1
1.2	Design Philosophy	2
1.3	API Version 2	3
1.4	API Version 3	3
1.5	Conventions	4
1.6	Terminology	4
<b>2</b>	<b>Initialization</b>	<b>6</b>
2.1	glutInit	6
2.2	glutInitWindowPosition, glutInitWindowSize	7
2.3	glutInitDisplayMode	7
<b>3</b>	<b>Beginning Event Processing</b>	<b>8</b>
3.1	glutMainLoop	8
<b>4</b>	<b>Window Management</b>	<b>8</b>
4.1	glutCreateWindow	9
4.2	glutCreateSubWindow	9
4.3	glutSetWindow, glutGetWindow	10
4.4	glutDestroyWindow	10
4.5	glutPostRedisplay	10
4.6	glutSwapBuffers	11
4.7	glutPositionWindow	11
4.8	glutReshapeWindow	11
4.9	glutFullScreen	12
4.10	glutPopWindow, glutPushWindow	12
4.11	glutShowWindow, glutHideWindow, glutIconifyWindow	13
4.12	glutSetWindowTitle, glutSetIconTitle	13
4.13	glutSetCursor	13
<b>5</b>	<b>Overlay Management</b>	<b>14</b>
5.1	glutEstablishOverlay	14
5.2	glutUseLayer	15
5.3	glutRemoveOverlay	15
5.4	glutPostOverlayRedisplay	16
5.5	glutShowOverlay, glutHideOverlay	16
<b>6</b>	<b>Menu Management</b>	<b>16</b>
6.1	glutCreateMenu	16
6.2	glutSetMenu, glutGetMenu	17
6.3	glutDestroyMenu	17
6.4	glutAddMenuEntry	17
6.5	glutAddSubMenu	18
6.6	glutChangeToMenuEntry	18
6.7	glutChangeToSubMenu	18
6.8	glutRemoveMenuItem	19
6.9	glutAttachMenu, glutDetachMenu	19

<b>7</b>	<b>Callback Registration</b>	<b>19</b>
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
7.12	glutSpaceballButtonFunc . . . . .	25
7.13	glutButtonBoxFunc . . . . .	26
7.14	glutDialsFunc . . . . .	26
7.15	glutTabletMotionFunc . . . . .	27
7.16	glutTabletButtonFunc . . . . .	27
7.17	glutMenuStatusFunc . . . . .	27
7.18	glutIdleFunc . . . . .	28
7.19	glutTimerFunc . . . . .	28
<b>8</b>	<b>Color Index Colormap Management</b>	<b>29</b>
8.1	glutSetColor . . . . .	29
8.2	glutGetColor . . . . .	29
8.3	glutCopyColormap . . . . .	30
<b>9</b>	<b>State Retrieval</b>	<b>30</b>
9.1	glutGet . . . . .	30
9.2	glutLayerGet . . . . .	32
9.3	glutDeviceGet . . . . .	32
9.4	glutGetModifiers . . . . .	33
9.5	glutExtensionSupported . . . . .	33
<b>10</b>	<b>Font Rendering</b>	<b>34</b>
10.1	glutBitmapCharacter . . . . .	34
10.2	glutBitmapWidth . . . . .	35
10.3	glutStrokeCharacter . . . . .	35
10.4	glutStrokeWidth . . . . .	36
<b>11</b>	<b>Geometric Object Rendering</b>	<b>36</b>
11.1	glutSolidSphere, glutWireSphere . . . . .	36
11.2	glutSolidCube, glutWireCube . . . . .	36
11.3	glutSolidCone, glutWireCone . . . . .	37
11.4	glutSolidTorus, glutWireTorus . . . . .	37
11.5	glutSolidDodecahedron, glutWireDodecahedron . . . . .	38
11.6	glutSolidOctahedron, glutWireOctahedron . . . . .	38
11.7	glutSolidTetrahedron, glutWireTetrahedron . . . . .	38
11.8	glutSolidIcosahedron, glutWireIcosahedron . . . . .	38
11.9	glutSolidTeapot, glutWireTeapot . . . . .	39
<b>12</b>	<b>Usage Advice</b>	<b>39</b>

<b>13 FORTRAN Binding</b>	<b>41</b>
13.1 Names for the FORTRAN GLUT Binding . . . . .	41
13.2 Font Naming Caveat . . . . .	41
13.3 NULL Callback . . . . .	42
<b>14 Implementation Issues</b>	<b>42</b>
14.1 Name Space Conventions . . . . .	42
14.2 Modular Implementation . . . . .	42
14.3 Error Checking and Reporting . . . . .	42
14.4 Avoid Unspecified GLUT Usage Restrictions . . . . .	42
<b>A GLUT State</b>	<b>44</b>
A.1 Types of State . . . . .	44
A.2 Global State . . . . .	44
A.3 Window State . . . . .	45
A.4 Menu State . . . . .	48
<b>B glut.h ANSI C Header File</b>	<b>49</b>
<b>C fglut.h FORTRAN Header File</b>	<b>55</b>
<b>References</b>	<b>60</b>
<b>Index</b>	<b>61</b>



# 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 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 ANSI C 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.

**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 pop-menu 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`- prefixed 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 *argc, char **argv);
```

`argc` A pointer to the program's *unmodified* `argc` variable from `main`. Upon return, the value pointed to by `argc` 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 `argc`, 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 *WxH+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.

- 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`

`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

```
int glutCreateSubWindow(int win,  
                        int x, int y, int width, int height);
```

`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`.

## 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's parent 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 *current 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*.

**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 *layer 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.

**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.

**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.

**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

```
void glutMouseFunc(void (*func)(int button, int state,
                                int x, int y));
```

`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.

**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*.

**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.

## 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

```
void glutTabletButtonFunc(void (*func)(int button, int state,  
                                     int x, int y));
```

`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. There is a single menu status callback 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**

```
void glutTimerFunc(unsigned int msec,
                   void (*func)(int value), value);
```

`msec` 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

```
void glutSetColor(int cell,
                  GLfloat red, GLfloat green, GLfloat blue);
```

`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*.

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

```
void glutSolidSphere(GLdouble radius,
                    GLint slices, GLint stacks);
void glutWireSphere(GLdouble radius,
                   GLint slices, GLint stacks);
```

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.

**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**

```
void glutSolidCone(GLdouble base, GLdouble height,
                  GLint slices, GLint stacks);
void glutWireCone(GLdouble base, GLdouble height,
                  GLint slices, GLint stacks);
```

*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  $Z = \text{height}$ . 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**

```
void glutSolidTorus(GLdouble innerRadius,
                  GLdouble outerRadius,
                  GLint nsides, GLint rings);
void glutWireTorus(GLdouble innerRadius,
                  GLdouble outerRadius,
                  GLint nsides, GLint rings);
```

*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 (20-sided regular solid) respectively.

**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 teapot<sup>1</sup> 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*.

---

<sup>1</sup> Yes, 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 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 `-glddebug` 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. FORTRAN 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 FORTRAN program would look like:

```
f77 -o foo foo.c -lfglut -lglut -lGLU -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 compilers.

### 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 outweigh 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.2.1 Program Controlled State

Name	Type	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.

### A.2.2 Fixed System Dependent State

Name	Type	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.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	Type	Set/Change	Get	Initial
number	Integer	-	glutGetWindow	<i>top-level</i> : glutCreateWindow (1) <i>sub-win</i> : glutCreateSubWindow (1)
x	Integer	glutPositionWindow	GLUT_WINDOW_X	<i>top-level</i> : initWindowX (2) <i>sub-win</i> : glutCreateSubWindow
y	Integer	glutPositionWindow	GLUT_WINDOW_Y	<i>top-level</i> : initWindowY (3) <i>sub-win</i> : glutCreateSubWindow
width	Integer	glutReshapeWindow	GLUT_WINDOW_WIDTH	<i>top-level</i> : initWindowWidth (4) <i>sub-win</i> : glutCreateSubWindow
height	Integer	glutReshapeWindow	GLUT_WINDOW_HEIGHT	<i>top-level</i> : initWindowHeight (5) <i>sub-win</i> : glutCreateSubWindow
top-level: fullScreen	Boolean	glutFullScreen glutPositionWindow glutReshapeWindow (6)		False
cursor	Cursor	glutSetCursor	GLUT_WINDOW_CURSOR	GLUT_CURSOR_INHERIT
stacking	Stacking	glutPopWindow glutPushWindow	-	top
displayState	State (7)	glutShowWindow (8) glutHideWindow glutIconifyWindow (9)	-	shown
visibility	Visibility	glutPostRedisplay (11)	(10)	undefined
redisplay	Boolean	glutWindowTitle	-	False
top-level: windowTitle	String	glutIconTitle	-	glutCreateWindow
top-level: iconTitle	String	glutDisplayFunc	-	glutCreateWindow
displayCallback	Callback	glutReshapeFunc	-	NULL (12)
reshapeCallback	Callback	glutKeyboardFunc	-	NULL (13)
keyboardCallback	Callback	glutMouseFunc	-	NULL
mouseCallback	Callback	glutMotionFunc	-	NULL
motionCallback	Callback	glutPassiveMotionFunc	-	NULL
passiveMotionCallback	Callback	glutSpecialFunc	-	NULL
specialCallback	Callback	glutSpaceballMotionFunc	-	NULL
spaceballMotionCallback	Callback	glutSpaceballRotateFunc	-	NULL
spaceballRotateCallback	Callback	glutSpaceballButtonFunc	-	NULL
spaceballButtonCallback	Callback	glutButtonBoxFunc	-	NULL
buttonBoxCallback	Callback	glutDialsFunc	-	NULL
dialsCallback	Callback	glutTabletMotionFunc	-	NULL
tabletMotionCallback	Callback	glutTabletButtonFunc	-	NULL
tabletButtonCallback	Callback	glutVisibilityFunc	-	NULL
visibilityCallback	Callback	glutEntryFunc	-	NULL
entryCallback	Callback	glutSetColor glutCopyColormap	glutGetColor	undefined
<i>cindex</i> : colormap	array of ColorCell	-	GLUT_WINDOW_PARENT	<i>top-level</i> : 0 <i>sub-win</i> : (14)
windowParent	Integer	glutCreateSubWindow glutDestroyWindow	GLUT_NUM_CHILDREN	0
numChildren	Integer	glutAttachMenu glutDetachMenu	-	0
leftMenu	Integer	glutAttachMenu glutDetachMenu	-	0
middleMenu	Integer	glutAttachMenu glutDetachMenu	-	0
rightMenu	Integer	glutAttachMenu glutDetachMenu	-	0

- (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`.

- (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	Type	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	Type	Set/Change	Get	Initial
hasOverlay	Boolean	<code>glutEstablishOverlay</code> <code>glutRemoveOverlay</code>	GLUT_HAS_OVERLAY	False
overlayPossible	Boolean	(1)	GLUT_OVERLAY_POSSIBLE	False
layerInUse	Layer	<code>glutUseLayer</code> (2)	GLUT_LAYER_IN_USE	normal plane
<i>cindex</i> : transparentIndex	Integer	-	GLUT_TRANSPARENT_INDEX	(3)
overlayRedisplay	Boolean	<code>glutPostOverlayRedisplay</code> (4)	-	False
overlayDisplayCallback	Callback	<code>glutOverlayDisplayFunc</code>	-	NULL
overlayDisplayState	State	<code>glutShowOverlay</code> <code>glutHideOverlay</code>	-	shown
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	Type	Set/Change	Get	Initial
number	Integer	-	glutSetMenu	<i>top-level</i> : 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

```

1  #ifndef __glut_h__
2  #define __glut_h__
3
4  /* Copyright (c) Mark J. Kilgard, 1994, 1995, 1996. */
5
6  /* This program is freely distributable without licensing fees and is
7     provided without guarantee or warranty expressed or implied. This
8     program is -not- in the public domain. */
9
10 #include <GL/gl.h>
11 #include <GL/glu.h>
12
13 #ifdef __cplusplus
14 extern "C" {
15 #endif
16
17 /*
18  * GLUT API revision history:
19  *
20  * GLUT_API_VERSION is updated to reflect incompatible GLUT
21  * API changes (interface changes, semantic changes, deletions,
22  * or additions).
23  *
24  * GLUT_API_VERSION=1  First public release of GLUT.  11/29/94
25  *
26  * GLUT_API_VERSION=2  Added support for OpenGL/GLX multisampling,
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 overridden */
34 #define GLUT_API_VERSION      3
35 #endif
36
37 /*
38  * GLUT implementation revision history:
39  *
40  * GLUT_XLIB_IMPLEMENTATION is updated to reflect both GLUT
41  * API revisions and implementation revisions (ie, bug fixes).
42  *
43  * GLUT_XLIB_IMPLEMENTATION=1  mjk's first public release of
44  * GLUT Xlib-based implementation.  11/29/94
45  *
46  * GLUT_XLIB_IMPLEMENTATION=2  mjk's second public release of
47  * GLUT Xlib-based implementation providing GLUT version 2
48  * interfaces.
49  *
50  * GLUT_XLIB_IMPLEMENTATION=3  mjk's GLUT 2.2 images.  4/17/95
51  *
52  * GLUT_XLIB_IMPLEMENTATION=4  mjk's GLUT 2.3 images.  6/?/95
53  *
54  * GLUT_XLIB_IMPLEMENTATION=5  mjk's GLUT 3.0 images.  10/?/95
55  *
56  * GLUT_XLIB_IMPLEMENTATION=6  mjk's GLUT 3.1
57  */
58 #ifndef GLUT_XLIB_IMPLEMENTATION /* allow this to be overridden */
59 #define GLUT_XLIB_IMPLEMENTATION      6
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 #define GLUT_ACCUM                  4
69 #define GLUT_ALPHA                   8
70 #define GLUT_DEPTH                   16
71 #define GLUT_STENCIL                 32
72 #if (GLUT_API_VERSION >= 2)
73 #define GLUT_MULTISAMPLE             128
74 #define GLUT_STEREO                 256
75 #endif
76 #if (GLUT_API_VERSION >= 3)
77 #define GLUT_LUMINANCE               512
78 #endif
79
80 /* mouse buttons */
81 #define GLUT_LEFT_BUTTON              0
82 #define GLUT_MIDDLE_BUTTON           1
83 #define GLUT_RIGHT_BUTTON            2
84
85 /* mouse button callback state */
86 #define GLUT_DOWN                     0
87 #define GLUT_UP                       1
88
89 #if (GLUT_API_VERSION >= 2)
90 /* function keys */
91 #define GLUT_KEY_F1                   1
92 #define GLUT_KEY_F2                   2
93 #define GLUT_KEY_F3                   3
94 #define GLUT_KEY_F4                   4
95 #define GLUT_KEY_F5                   5
96 #define GLUT_KEY_F6                   6
97 #define GLUT_KEY_F7                   7
98 #define GLUT_KEY_F8                   8
99 #define GLUT_KEY_F9                   9
100 #define GLUT_KEY_F10                  10
101 #define GLUT_KEY_F11                  11
102 #define GLUT_KEY_F12                  12
103 /* directional keys */
104 #define GLUT_KEY_LEFT                 100
105 #define GLUT_KEY_UP                   101
106 #define GLUT_KEY_RIGHT                102
107 #define GLUT_KEY_DOWN                 103
108 #define GLUT_KEY_PAGE_UP              104
109 #define 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
115 /* entry/exit callback state */
116 #define GLUT_LEFT                      0
117 #define GLUT_ENTERED                  1
118
119 /* menu usage callback state */
120 #define GLUT_MENU_NOT_IN_USE           0
121 #define GLUT_MENU_IN_USE              1
122
123 /* visibility callback state */
124 #define GLUT_NOT_VISIBLE               0
125 #define GLUT_VISIBLE                  1
126
127 /* color index component selection values */
128 #define GLUT_RED                       0
129 #define GLUT_GREEN                     1
130 #define GLUT_BLUE                      2
131
132 /* layers for use */
133 #define GLUT_NORMAL                    0
134 #define GLUT_OVERLAY                   1

```

```

135
136 /* stroke font opaque addresses (use constants instead in source code) */
137 extern void *glutStrokeRoman;
138 extern void *glutStrokeMonoRoman;
139
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 *glutBitmapHelvetica12;
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

```

```

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
213 #define GLUT_HAS_TABLET            604
214 #define GLUT_NUM_MOUSE_BUTTONS     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

```

```

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
396 #endif          /* __glut_h__ */

```

## 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
11 parameter ( GLUT_RGB = 0 )
12 integer*4 GLUT_RGBA
13 parameter ( GLUT_RGBA = 0 )
14 integer*4 GLUT_INDEX
15 parameter ( GLUT_INDEX = 1 )
16 integer*4 GLUT_SINGLE
17 parameter ( GLUT_SINGLE = 0 )
18 integer*4 GLUT_DOUBLE
19 parameter ( GLUT_DOUBLE = 2 )
20 integer*4 GLUT_ACCUM
21 parameter ( GLUT_ACCUM = 4 )
22 integer*4 GLUT_ALPHA
23 parameter ( GLUT_ALPHA = 8 )
24 integer*4 GLUT_DEPTH
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
34 integer*4 GLUT_LEFT_BUTTON
35 parameter ( GLUT_LEFT_BUTTON = 0 )
36 integer*4 GLUT_MIDDLE_BUTTON
37 parameter ( GLUT_MIDDLE_BUTTON = 1 )
38 integer*4 GLUT_RIGHT_BUTTON
39 parameter ( GLUT_RIGHT_BUTTON = 2 )
40
41 C mouse button callback state
42 integer*4 GLUT_DOWN
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 )
58 integer*4 GLUT_KEY_F6
59 parameter ( GLUT_KEY_F6 = 6 )
60 integer*4 GLUT_KEY_F7
61 parameter ( GLUT_KEY_F7 = 7 )
62 integer*4 GLUT_KEY_F8
63 parameter ( GLUT_KEY_F8 = 8 )
64 integer*4 GLUT_KEY_F9
65 parameter ( GLUT_KEY_F9 = 9 )
66 integer*4 GLUT_KEY_F10

```

```

67     parameter ( GLUT_KEY_F10 = 10 )
68     integer*4 GLUT_KEY_F11
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 )
74     integer*4 GLUT_KEY_UP
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 )
84     integer*4 GLUT_KEY_HOME
85     parameter ( GLUT_KEY_HOME = 106 )
86     integer*4 GLUT_KEY_END
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
92     integer*4 GLUT_LEFT
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
137 C  glutGet parameters
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
147      parameter ( GLUT_WINDOW_BUFFER_SIZE = 104 )
148      integer*4 GLUT_WINDOW_STENCIL_SIZE
149      parameter ( GLUT_WINDOW_STENCIL_SIZE = 105 )
150      integer*4 GLUT_WINDOW_DEPTH_SIZE
151      parameter ( GLUT_WINDOW_DEPTH_SIZE = 106 )
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
159      parameter ( GLUT_WINDOW_ALPHA_SIZE = 110 )
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 )
202      integer*4 GLUT_INIT_WINDOW_HEIGHT

```

```

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
219     parameter ( GLUT_HAS_TABLET = 604 )
220     integer*4 GLUT_NUM_MOUSE_BUTTONS
221     parameter ( GLUT_NUM_MOUSE_BUTTONS = 605 )
222     integer*4 GLUT_NUM_SPACEBALL_BUTTONS
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
232     integer*4 GLUT_OVERLAY_POSSIBLE
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
237     parameter ( GLUT_HAS_OVERLAY = 802 )
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 )
256     integer*4 GLUT_ACTIVE_ALT
257     parameter ( GLUT_ACTIVE_ALT = 4 )
258
259 C   glutSetCursor parameters
260     integer*4 GLUT_CURSOR_RIGHT_ARROW
261     parameter ( GLUT_CURSOR_RIGHT_ARROW = 0 )
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 )
270     integer*4 GLUT_CURSOR_CYCLE

```

```
271     parameter ( GLUT_CURSOR_CYCLE = 5 )
272     integer*4 GLUT_CURSOR_SPRAY
273     parameter ( GLUT_CURSOR_SPRAY = 6 )
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
305     parameter ( GLUT_CURSOR_FULL_CROSSHAIR = 102 )
306
307 C GLUT functions
308     integer*4 glutcreatewindow
309     integer*4 glutgetwindow
310     integer*4 glutcreatemenu
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 OpenGL™ 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  
`_SGL_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

OpenGL<sup>®</sup> Graphics with the X Window System<sup>®</sup>  
(Version 1.3)

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

<b>1</b>	<b>Overview</b>	<b>1</b>
<b>2</b>	<b>GLX Operation</b>	<b>2</b>
2.1	Rendering Contexts and Drawing Surfaces . . . . .	2
2.2	Using Rendering Contexts . . . . .	3
2.3	Direct Rendering and Address Spaces . . . . .	4
2.4	OpenGL Display Lists . . . . .	5
2.5	Texture Objects . . . . .	6
2.6	Aligning Multiple Drawables . . . . .	7
2.7	Multiple Threads . . . . .	7
<b>3</b>	<b>Functions and Errors</b>	<b>9</b>
3.1	Errors . . . . .	9
3.2	Events . . . . .	10
3.3	Functions . . . . .	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	Off Screen Rendering . . . . .	21
3.3.6	Querying Attributes . . . . .	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	Backwards Compatibility . . . . .	35
3.4.1	Using Visuals for Configuration Management . . . . .	35
3.4.2	Off Screen Rendering . . . . .	39

3.5	Rendering Contexts . . . . .	40
<b>4</b>	<b>Encoding on the X Byte Stream</b>	<b>42</b>
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
<b>5</b>	<b>Extending OpenGL</b>	<b>47</b>
<b>6</b>	<b>GLX Versions</b>	<b>49</b>
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
<b>7</b>	<b>Glossary</b>	<b>51</b>
	<b>Index of GLX Commands</b>	<b>53</b>

# 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 <code>glXChooseVisual</code> . . . . .	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 **Pixmap**s 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. **GLXPbuffer**s are used for offscreen rendering but they have different semantics than **GLXPixmap**s that make it easier to allocate them in non-visible frame buffer memory.

**GLXWindows**, **GLXPixmap**s and **GLXPbuffer**s are created with respect to a **GLXFBCConfig**; the **GLXFBCConfig** 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 **GLXFBCConfig**.

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`.

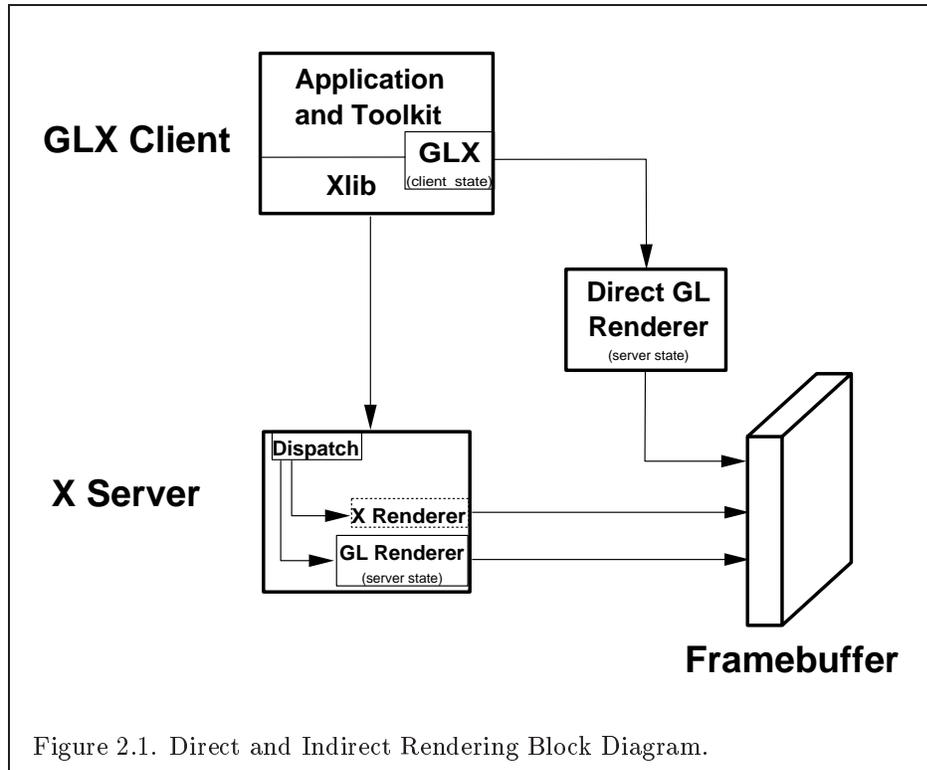
For backwards compatibility with GLX versions 1.2 and earlier, a rendering context can also be used to render into a `Window`. Thus, a `GLXDrawable` is the union `{GLXWindow, GLXPixmap, GLXPbuffer, Window}`. In X, `Windows` are associated with a `Visual`. In GLX the definition of `Visual` has been extended 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 **glBindTexture** are atomic. Also, a texture object will not be deleted until it is no longer bound to any rendering context.

## 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

```
Bool glXQueryExtension(Display *dpy, int
    *error_base, int *event_base);
```

*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:

```
const char *glXQueryExtensionsString(Display *dpy,
    int screen);
```

`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-separated 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 **glXGetClientString**. 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 **GLXPixmap**s and **GLXPbuffer**s 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_STEREO	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: 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 `GLXFBConfig` for which the value of the `GLX_DRAWABLE_TYPE` 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 will not pass the required OpenGL conformance tests.

Servers are required to export at least one `GLXFBCConfig` 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 `GLXFBCConfig` 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; auxiliary buffers are optional, and the alpha buffer may have 0 bits. The color buffer size for this `GLXFBCConfig` 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 `GLXFBCConfig` 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 `GLXFBCConfig` 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`.

If `GLX_TRANSPARENT_TYPE` is `GLX_NONE` or `GLX_TRANSPARENT_RGB`, then the value for `GLX_TRANSPARENT_INDEX_VALUE` is undefined. If `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_RED_VALUE`, `GLX_TRANSPARENT_GREEN_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 GLXFBConfigs 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 and Sorting Criteria	Sort Priority
GLX_FBCONFIG_ID	GLX_DONT_CARE	<i>Exact</i>	
GLX_BUFFER_SIZE	0	<i>Smaller</i>	3
GLX_LEVEL	0	<i>Exact</i>	
GLX_DOUBLEBUFFER	GLX_DONT_CARE	<i>Exact</i>	4
GLX_STEREO	False	<i>Exact</i>	
GLX_AUX_BUFFERS	0	<i>Smaller</i>	5
GLX_RED_SIZE	0	<i>Larger</i>	2
GLX_GREEN_SIZE	0	<i>Larger</i>	2
GLX_BLUE_SIZE	0	<i>Larger</i>	2
GLX_ALPHA_SIZE	0	<i>Larger</i>	2
GLX_DEPTH_SIZE	0	<i>Larger</i>	6
GLX_STENCIL_SIZE	0	<i>Larger</i>	7
GLX_ACCUM_RED_SIZE	0	<i>Larger</i>	8
GLX_ACCUM_GREEN_SIZE	0	<i>Larger</i>	8
GLX_ACCUM_BLUE_SIZE	0	<i>Larger</i>	8
GLX_ACCUM_ALPHA_SIZE	0	<i>Larger</i>	8
GLX_RENDER_TYPE	GLX_RGBA_BIT	<i>Mask</i>	
GLX_DRAWABLE_TYPE	GLX_WINDOW_BIT	<i>Mask</i>	
GLX_X_RENDERABLE	GLX_DONT_CARE	<i>Exact</i>	
GLX_X_VISUAL_TYPE	GLX_DONT_CARE	<i>Exact</i>	9
GLX_CONFIG_CAVEAT	GLX_DONT_CARE	<i>Exact</i>	1
GLX_TRANSPARENT_TYPE	GLX_NONE	<i>Exact</i>	
GLX_TRANSPARENT_INDEX_VALUE	GLX_DONT_CARE	<i>Exact</i>	
GLX_TRANSPARENT_RED_VALUE	GLX_DONT_CARE	<i>Exact</i>	
GLX_TRANSPARENT_GREEN_VALUE	GLX_DONT_CARE	<i>Exact</i>	
GLX_TRANSPARENT_BLUE_VALUE	GLX_DONT_CARE	<i>Exact</i>	
GLX_TRANSPARENT_ALPHA_VALUE	GLX_DONT_CARE	<i>Exact</i>	

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

```
int glXGetFBConfigAttrib(Display *dpy, GLXFBConfig
    config, int attribute, int *value);
```

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 `GLXPixmap`s 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

```
GLXPixmap glXCreatePixmap(Display *dpy, GLXFBConfig
    config, Pixmap pixmap, const int *attrib_list);
```

`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, `glXSwapBuffers` 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 `GLXFBCconfig` then a `GLXBadFBCconfig` 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:

```
void glXQueryDrawable(Display *dpy, GLXDrawable
    draw, int attribute, unsigned int *value);
```

`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` puffers.

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

```
void glXDestroyContext(Display *dpy, GLXContext
    ctx);
```

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 `GLXWindow` or `GLXPbuffer` and *ctx* was previously bound to 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 `glReadBuffer` (even if they are implicitly called via `glPopAttrib` or `glXCopyContext`) 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

Table 3.5: Context attributes.

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

```
int glXQueryContext(Display *dpy, GLXContext ctx,
    int attribute, int *value);
```

**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 GLXWindow or GLXPbuffer call

```
void glXGetSelectedEvent(Display *dpy, GLXDrawable
    draw, unsigned long *event_mask);
```

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

```
void glXUseXFont(Font font, int first, int count,
    int list_base);
```

*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.

```
int glXGetConfig(Display *dpy, XVisualInfo *visual,
                int attribute, int *value);
```

**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	<i>Exact</i>
GLX_BUFFER_SIZE	0	<i>Smaller</i>
GLX_LEVEL	0	<i>Exact</i>
GLX_RGBA	False	<i>Exact</i>
GLX_DOUBLEBUFFER	False	<i>Exact</i>
GLX_STEREO	False	<i>Exact</i>
GLX_AUX_BUFFERS	0	<i>Smaller</i>
GLX_RED_SIZE	0	<i>Larger</i>
GLX_GREEN_SIZE	0	<i>Larger</i>
GLX_BLUE_SIZE	0	<i>Larger</i>
GLX_ALPHA_SIZE	0	<i>Larger</i>
GLX_DEPTH_SIZE	0	<i>Larger</i>
GLX_STENCIL_SIZE	0	<i>Smaller</i>
GLX_ACCUM_RED_SIZE	0	<i>Larger</i>
GLX_ACCUM_GREEN_SIZE	0	<i>Larger</i>
GLX_ACCUM_BLUE_SIZE	0	<i>Larger</i>
GLX_ACCUM_ALPHA_SIZE	0	<i>Larger</i>

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, `GLXPixmap`s 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 `glXMakeContextCurrent(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

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

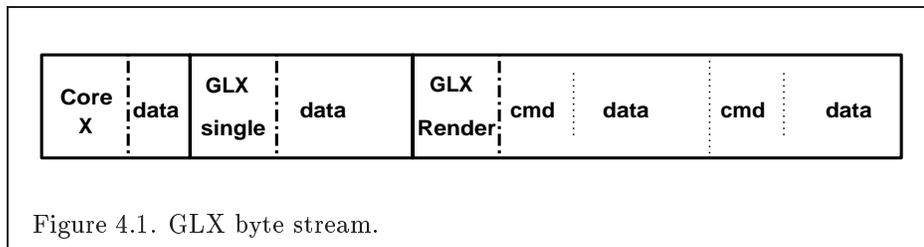
# Encoding on the X Byte Stream

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 `<glx.h>` (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
```



**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**  
**glXMakeContextCurrent**  
**glXMakeCurrent**  
**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 **glXVendorPrivateWithReply** request is required; otherwise protocol for a new rendering command that can be sent to the X Server as part of a **glXRender** or **glXRenderLarge** request is required.

The OpenGL Architectural Review Board maintains a registry of vendor-specific enumerated values; opcodes for vendor private requests, vendor private with reply requests, and OpenGL rendering commands; and vendor-specific 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 **glXVendorPrivate-**

**WithReply** 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`  
`glXGetFBConfigAttrib`  
`glXGetVisualFromFBConfig`  
`glXCreateWindow`  
`glXDestroyWindow`  
`glXCreatePixmap`  
`glXDestroyPixmap`  
`glXCreatePbuffer`  
`glXDestroyPbuffer`  
`glXQueryDrawable`  
`glXCreateNewContext`  
`glXMakeContextCurrent`  
`glXGetCurrentReadDrawable`  
`glXQueryContext`  
`glXSelectEvent`  
`glXGetSelectedEvent`

## 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 `GLXPixmap`s 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

- GLX\_BACK\_RIGHT\_BUFFER\_BIT, 32
- GLX\_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, 38
- GLX\_COLOR\_INDEX\_BIT, 12, 15
- GLX\_COLOR\_INDEX\_TYPE, 25, 40
- GLX\_CONFIG\_CAVEAT, 13, 15, 18, 19
- GLX\_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, 18-21
- GLX\_EXTENSIONS, 11
- GLX\_FBCONFIG\_ID, 13, 18, 19, 25, 30, 36, 39, 40
- GLX\_FRONT\_LEFT\_BUFFER\_BIT, 32
- GLX\_FRONT\_RIGHT\_BUFFER\_BIT, 32
- GLX\_GRAY\_SCALE, 14, 20
- GLX\_GREEN\_SIZE, 12, 13, 18, 19, 36, 38
- GLX\_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, 23-25
- GLX\_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

- 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, 18–20
- GLXBadContext, 9, 26, 27, 29, 30, 40
- GLXBadContextState, 9, 27, 35, 40
- GLXBadContextTag, 10
- GLXBadCurrentDrawable, 9, 27, 29, 33–35
- GLXBadCurrentWindow, 9, 40
- GLXBadDrawable, 9, 25, 27, 31, 34, 40
- GLXBadFBConfig, 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, 50
- glXCreatePbuffer, 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, 50
- glXGetFBConfigAttrib, 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

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

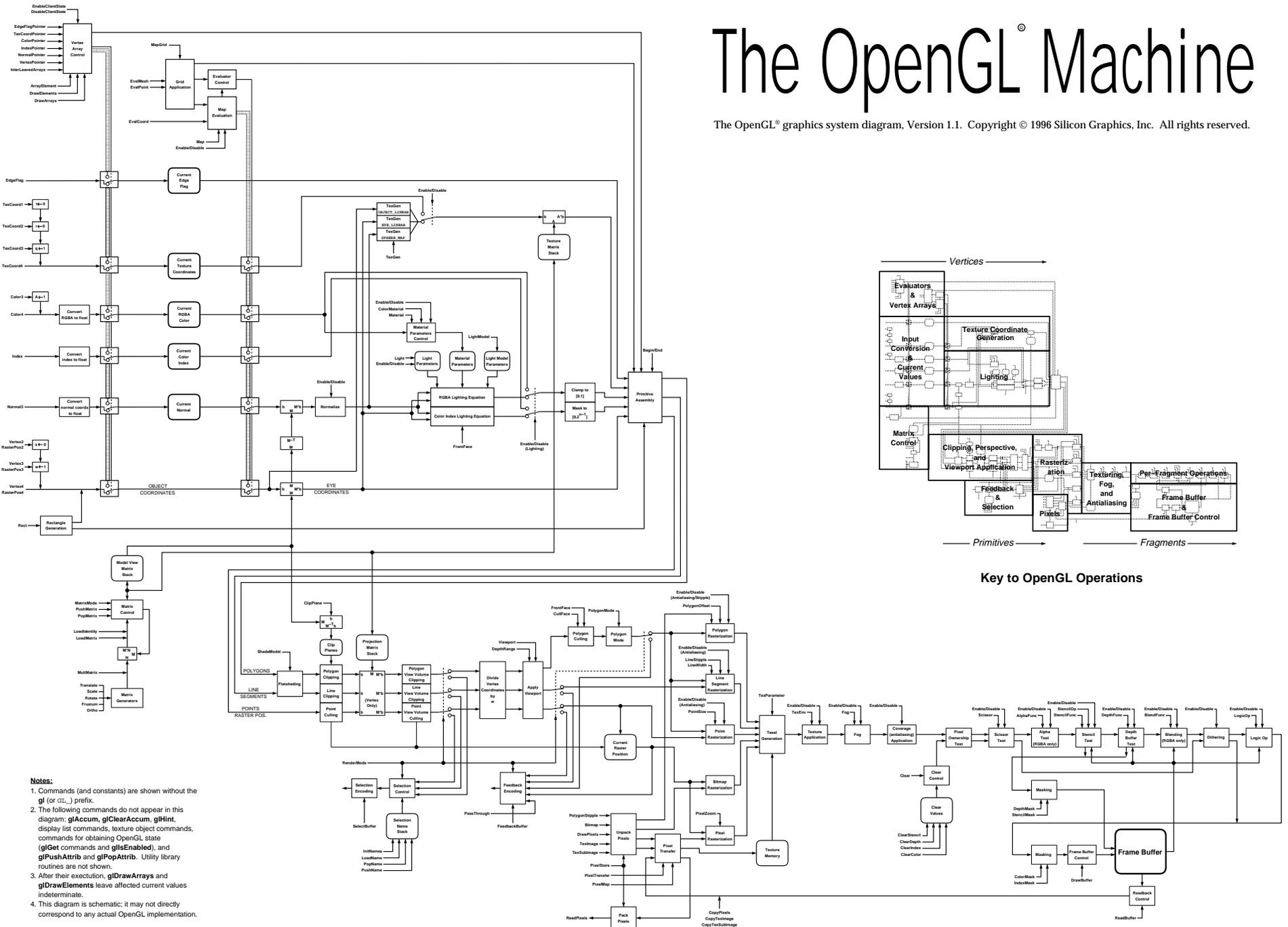
XFreePixmap, 23

XSync, 8, 24, 33

XVisualInfo, 35

# The OpenGL<sup>®</sup> Machine

The OpenGL<sup>®</sup> graphics system diagram, Version 1.1. Copyright © 1996 Silicon Graphics, Inc. All rights reserved.



- Notes:**
1. Commands (and constants) are shown without the `gl` (or `GL_`) prefix.
  2. The following commands do not appear in this diagram: `glAccum`, `glClearAccum`, `glHint`, display list commands, texture object commands, commands for obtaining OpenGL state (`glGet` commands and `glIsEnabled`), and `glPushAttrib` and `glPopAttrib`. Utility library routines are not shown.
  3. After their execution, `glDrawArrays` and `glDrawElements` leave affected current values indeterminate.
  4. This diagram is schematic; it may not directly correspond to any actual OpenGL implementation.

# SIGGRAPH '97

## Course 24: OpenGL and Window System Integration

### OpenGL Performance Optimization

#### Contents

- [1. Hardware vs. Software](#)
- [2. Application Organization](#)
  - [2.1 High Level Organization](#)
  - [2.2 Low Level Organization](#)
- [3. OpenGL Optimization](#)
  - [3.1 Traversal](#)
  - [3.2 Transformation](#)
  - [3.3 Rasterization](#)
  - [3.4 Texturing](#)
  - [3.5 Clearing](#)
  - [3.6 Miscellaneous](#)
  - [3.7 Window System Integration](#)
  - [3.8 Mesa-specific](#)
- [4. Evaluation and tuning](#)
  - [4.1 Pipeline tuning](#)
  - [4.2 Double buffering](#)
  - [4.3 Test on several implementations](#)

## 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 cheap 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 tessellation during animation. Use `glPolygonMode( GL_FRONT_AND_BACK, GL_LINE )` to inspect tessellation 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 {
```

## OpenGL Performance Optimization

```
        float latitude, 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[] )
{
    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++) {
        glRasterPos2f( citylist[i].longitude, citylist[i].latitude );
        glCallLists( strlen(citylist[i].name),
                    GL_BYTE,
                    citylist[i].name );
    }
}
```

## OpenGL Performance Optimization

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:

```
struct city_list {
    int num_cities;           /* how many cities in the list */
    float *position;        /* pointer to lat/lon coordinates */
    char **name;            /* pointer to city names */
    float size;             /* size of city points */
};
```

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 compromise 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 `glGet*` 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. tessellation 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) {
```

## OpenGL Performance Optimization

```
        glVertex3fv( 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++) {
        glVertex3fv( norm[i] );
        glVertex3fv( vert[i] );
    }
    glEnd();
}
else {
    glBegin( GL_TRIANGLE_STRIP );
    for (i=0; i < n ;i++) {
        glVertex3fv( vert[i] );
    }
    glEnd();
}
```

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

Connected primitives such as `GL_LINES`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, and `GL_QUAD_STRIP` decrease traversal and transformation load.

#### `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

## OpenGL Performance Optimization

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 performance improvement.

## OpenGL Performance Optimization

### 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 negligible.

### 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
```

## OpenGL Performance Optimization

```
* 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  $d \neq 1.0$

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

## OpenGL Performance Optimization

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 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 minimize 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 `glXWaitGL` to synchronize, then call all the X drawing functions.

Don't redraw more than necessary

Be sure that you're not redrawing your scene unnecessarily. 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 noticeable delay between mouse input and screen updates.

## OpenGL Performance Optimization

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

## OpenGL Performance Optimization

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 isolate the CPU subsystem one must reduce the graphics workload while preserving 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 isolate 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 achieve 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.



# Avoiding 16 Common OpenGL Pitfalls

**Mark J. Kilgard**  
[mjk@nvidia.com](mailto:mjk@nvidia.com)  
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 man 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.

## Avoiding 16 Common OpenGL Pitfalls

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. `glScalef` 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 to 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.

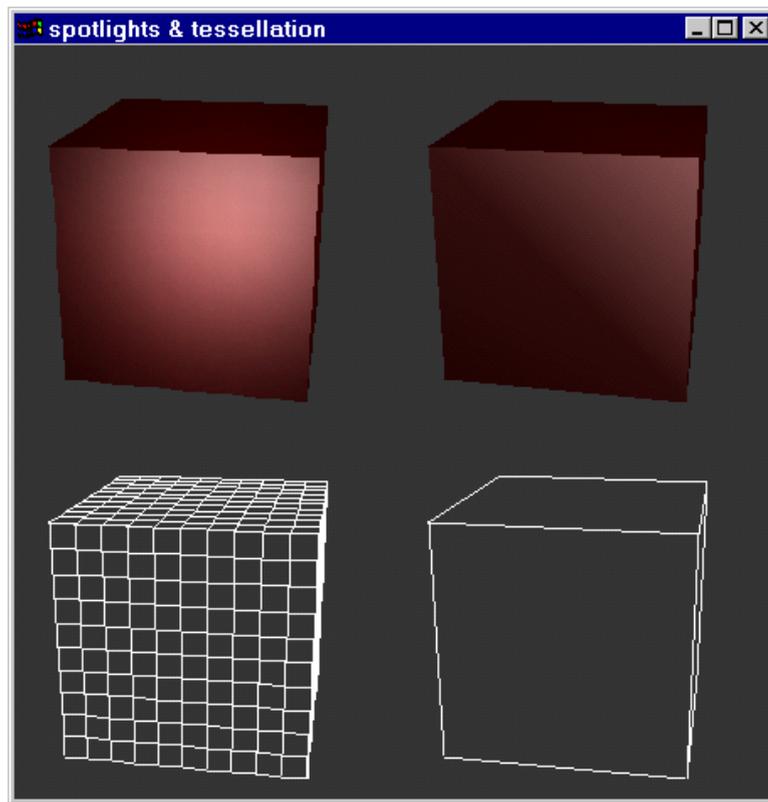
The other option is to adjust the normal vectors passed to OpenGL so that after the inverse transpose modelview transformation, the resulting normal will become a unit vector. For example, if the earlier `glScalef` call tripled the vertex coordinates, we could correct for this corresponding thirding 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's 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.

## Avoiding 16 Common OpenGL Pitfalls

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 `glMatrixMode` 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 newHeight)
{
    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

## Avoiding 16 Common OpenGL Pitfalls

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 `glGetIntegerv` 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.

## Avoiding 16 Common OpenGL Pitfalls

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 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 your 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 your 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);
```

## Avoiding 16 Common OpenGL Pitfalls

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 be 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);
```

## Avoiding 16 Common OpenGL Pitfalls

```
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 your 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).

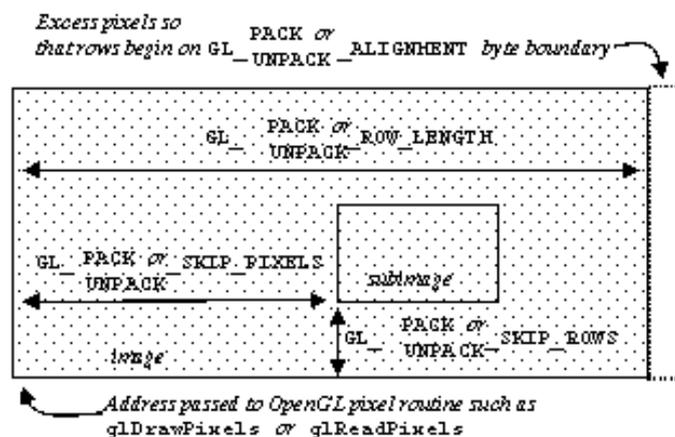


Figure 2: Relationship of the image layout pixel store modes.

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.

## Avoiding 16 Common OpenGL Pitfalls

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 \* Width \* Height*, but instead  $((\text{BytesPerPixel} * \text{Width} + 3) \gg 2) \ll 2 * \text{Height}$ . 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);
```

## Avoiding 16 Common OpenGL Pitfalls

```
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

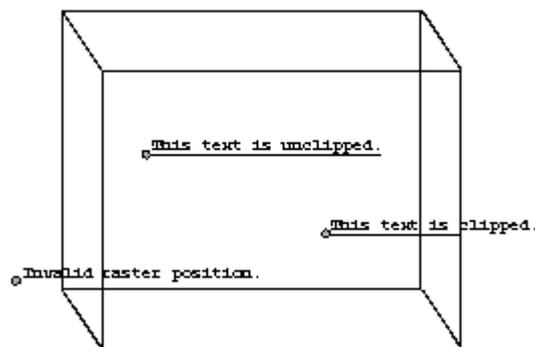
OpenGL's raster position determines where pixel rectangles and bitmaps will be rasterized. The `glRasterPos2f` family

## Avoiding 16 Common OpenGL Pitfalls

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, OpenGL's 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 `glCopyPixels` 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.**

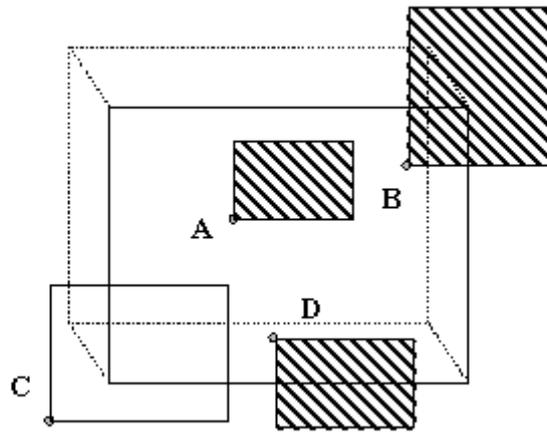
Consider how this can surprise you. Say you wanted to draw a string of text with each character rendered with `glBitmap`. Figure 3 shows a few situations. The point to notice is that the text renders as expected in the first two cases, but in the last case, the raster position's placement is outside 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 shown (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 within 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's viewport.

Part of the confusion comes from the fact that, most of the time, the viewport is set to be the window's rectangular extent and pixels are clipped to the window's 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. OpenGL's 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` call's 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 window's 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 off the screen, that is not what happens!

## Avoiding 16 Common OpenGL Pitfalls

Keep in mind that the extra pixels around a wide or antialiased point are generated at rasterization time, but if the point's 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 point's edge can be up to four pixels (half of 8.0) from the point's center and still generate fragments within the window's 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 they 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`'s 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 left-hand 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

## Avoiding 16 Common OpenGL Pitfalls

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 system's 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 window's 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 3<sup>rd</sup> and 4<sup>th</sup> 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 window's 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;
```

## Avoiding 16 Common OpenGL Pitfalls

```
        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);
    glVertex3f(1.0, 0.0, 0.0);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, d2);
    glVertex3f(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);
```

## Avoiding 16 Common OpenGL Pitfalls

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 OpenGL's 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 OpenGL's 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 application's 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 your 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 your 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.*

# 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](mailto: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: Brett Johnson, Paul Martz, Samuel Paik, 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, Pierre Alliez, Brian Bailey, Steve Baker, Lars Blaabjerg, Jeff Burrell, Angus Dorbie, Bob Ellison, Glenn Forney, Ron Fosner, Phil Frisbie Jr, Michael I. Gold, Paul Groves, Charles E. Hardwidge, Chris Hecker, Scott Heiman, Helios, Blaine Hodge, Mark Kilgard, Michael Kurth, Bruce Lamming, Jon Leech, Mike Lischke, Ben Loftin, Jean-Luc Martinez, Steve McAndrewSmith, Phil McRevis, Reed Mideke, Teri Morrison, Duncan Murdoch, Joel Parris, Lev Povalahev, Stephane Routelous, Schneide, Shaleh, Dave Shreiner, Andrew F. Vesper, Jon White, and Lucian Wischik.

## 1.030 Download the entire FAQ as a Zip file

Download the entire FAQ in a single [zip file](#) (~125KB).

## 1.040 Change Log

Date	Notes
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 <code>index.htm</code> , 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 [comp.graphics.algorithms FAQ](#) contains a lot of 3D graphics information that isn't specific to OpenGL.

An excellent general computer graphics text is *Computer Graphics: Principles and Practice*, Second Edition, by James Foley, et al. ISBN 0-201-12110-7. It has been mentioned that this book is out of print and that a third edition is planned for release in January 2001. However, some online book retailers still seem to have it for sale. Try [amazon.com](#).

[Delphi code for performing basic vector, matrix, and quaternion operations can be found here.](#)

### 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](#).

[OpenGL Basics FAQ](#)

[OpenGL Game Developer's FAQ](#)

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](#)

[The OpenGL.org web site](#) has the current OpenGL specification and manual pages. You can view the OpenGL spec v1.1 [online as a Web page](#).

[A repository of OpenGL implementations for several platforms](#)

[The GLUT source code distribution](#) 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 <http://www.wischik.com/lu/programmer/wingl.html> contains excellent information on Microsoft Windows OpenGL, especially with 3dfx hardware.

[The NeHe Web page](#) 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 of these books described 1.0 and 1.1, respectively.

The [OpenGL red book is online](#). However, links to online copies of this book don't seem to last long. [Reference pages similar to the OpenGL blue book are also online](#).

In addition to the red and blue books, the green and white books are for X Windows and Microsoft Windows programming, respectively. You can obtain a more exhaustive list of OpenGL books by visiting the [www.opengl.org](http://www.opengl.org) 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 Are there OpenGL implementations that come with source code?**

The [Mesa library](#) 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.

For information on using OpenGL through Borland C++ Builder, visit [Scott Heiman's Web page](#).

Here are three sites with info on using 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>.

[Information on using OpenGL from Delphi can be found here](#), and also at the [Delphi3D web page](#).

### **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 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 [links to OpenGL developer and end-user files](#). 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. [More info on the SGI libraries can be found here.](#) 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:

[Mesa](#) is a free OpenGL-like library that is available on a number of platforms. You might also check the Developer section at [The OpenGL Organization's Web page](#) 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 [question 2.060 above](#) 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 <code>gl.h</code> , <code>glu.h</code> , or <code>glut.h</code>  A GLUT source file should: <code>#include &lt;GL/glut.h&gt;</code> Non-GLUT source files should: <code>#include &lt;GL/glu.h&gt;</code> <code>#include &lt;GL/gl.h&gt;</code>
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 <code>windows.h</code> or included it after <code>gl.h</code> .  Source files that use neither GLUT nor MFC, but which make calls to OpenGL, should: <code>#include &lt;windows.h&gt;</code> <code>#include &lt;GL/gl.h&gt;</code>

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 __imp__glMatrixMode@4 file.obj : error LNK2001: unresolved external symbol __imp__glViewport@16 file.obj : error LNK2001: unresolved external symbol __imp__glLoadIdentity@0	Didn't link with opengl32.lib, glu32.lib, or glut32.lib.  <a href="#">Section 2.060 above</a> 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 <a href="#">section 2.060 above</a> 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->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.iPixelFormat = 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, gluPerspect
}

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, glRo
    // 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 failure.

### 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. <http://www.metrolink.com/>

[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 [Mesa](#) library also supports 1.2. The [OpenGL Sample Implementation](#) 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 <http://www.opengl.org>. The [red and blue books](#) have recently been revised to cover OpenGL 1.2 functionality.

## 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: <http://reality.sgi.com/mjk/glut3/glut-faq.html>.

### 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 display lists?
- 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 (GLUT_RIGHT);
    }
}

```

### 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 [download the GLUT source](#) 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.

Normally, 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 an event indicating the program is exiting.

For simple resources such as memory deallocation, this should not be a problem. The OS will free any memory that was in use by the process.

Of greater concern is prompting the user to save work or flushing data held in software buffers to files.

Unfortunately, GLUT doesn't provide a mechanism to handle such events, because the OS or window manager is killing the process in a forceful way.

One option is to hack the GLUT source, and disable the Close Window icon in the window frame title bar. This is distasteful, for it means you must now include the entire GLUT window creation code in your application.

In short, there is no good answer to this problem. Users, in general, should not expect good results when they exit an application in this way.

**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 from glutMainLoop() to your program, there is only one way to do this. You need to download the GLUT source and hack gluMainLoop() to do what you want it to. Then you need to compile and link into your program this hacked version of glutMainLoop().

**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 [red and blue books](#) and the GLU specification, which you can obtain from [the OpenGL.org Web page](#).

### 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 [the Mesa source](#), 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.

### 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`. ([Download the GLUT distribution](#)).

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 GLdouble 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?

[See Blaine Hodge's web page](#). Be aware that this page shows usage of [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)? ([How do I do that?](#))
- 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 temporarily disable hardware acceleration, 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 `glGetString(GL_VENDOR)` returns something other than "Microsoft Corporation", then you are using the board's ICD. If it returns "Microsoft Corporation", normally this implies you chose a pixel format that can't be accelerated by your device. However, this is also returned if your device has an MCD instead of an ICD, so 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, you know that you were hardware accelerated before. Don't forget to put the ICD back the way it was.

You can also gather performance data by rendering into the back buffer and comparing the results against known performance statistics for your device. [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 [Reactor Critical](#) 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." You can get it from <http://www.glsetup.com>.

#### **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?**

This won't work on some drivers, and is therefore unportable. I don't recommend it.

#### **5.080 Can I use use DirectDraw to change the screen resolution or desktop pixel depth?**

This probably depends on your graphics device and what, if any, support for this type of operation your device driver provides. You need to keep this basic tenet in mind: Microsoft doesn't require, and consequently does not test for, the ability to render OpenGL into a DirectDraw surface. For this reason, you shouldn't expect this window creation strategy to be available.

#### **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 `OnEraseBackground()` event handler, which should do nothing other than return `TRUE`. This tells Windows that you've cleared the window. Of course, you didn't really clear the window. However, returning `TRUE` keeps Microsoft from trying to do it for you, and should prevent flashing.

#### **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](#).

## 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);
```

## 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 [projection abuse](#). 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. Initial

/* 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:

```

gluLookAt(camera[0], camera[1], camera[2], /* look from camera XYZ */
          0, 0, 0, /* look at the origin */
          0, 1, 0); /* positive Y up vector */
glRotatef(orbitDegrees, 0.f, 1.f, 0.f); /* orbit the Y axis */
/* ...where orbitDegrees is derived from mouse motion */

glCallList(SCENE); /* draw the scene */

```

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.)

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) windowHeight / windowWidth;

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:

```

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(50.0, 1.0, 3.0, 7.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.0, 0.0, 5.0,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);

```

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 [question 8.030](#) for an explanation of this problem.

# 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 [review of matrix concepts](#) and his [article on Euler angles](#).

[Delphi code for performing basic vector, matrix, and quaternion operations can be found here.](#)

## 9.005 Are OpenGL matrices column-major or row-major?

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.

For programming purposes, OpenGL matrices are 16-value arrays with base vectors laid out contiguously in memory.

## 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.

## 9.020 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()`. [More information is available on `gluLookAt\(\)`.](#)

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.

More detailed information on using OpenGL as a rasterization-only API is in the [OpenGL Game Developer's FAQ](#).

### 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 [OpenGL Game Developer’s FAQ](#) 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 [projection shadows](#), 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:

$$\text{fov} * 0.5 = \arctan ((\text{top} - \text{bottom}) * 0.5 / \text{near})$$

Since  $\text{bottom} == -\text{top}$  for the symmetrical projection that `gluPerspective()` produces, then:

$$\begin{aligned} \text{top} &= \tan(\text{fov} * 0.5) * \text{near} \\ \text{bottom} &= -\text{top} \end{aligned}$$

The *left* and *right* parameters are simply functions of the *top*, *bottom*, and *aspect*:

$$\begin{aligned} \text{left} &= \text{aspect} * \text{bottom} \\ \text{right} &= \text{aspect} * \text{top} \end{aligned}$$

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 ();
glMatrixMode (GL_MODELVIEW);
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 ( $0 \leq Z \leq 1$ ).

#### **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. [See this example code](#) 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.

### 9.170 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.

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:

$$X_{ndc} = X_{cc}/W_{cc}$$

$$Y_{ndc} = Y_{cc}/W_{cc}$$

$$Z_{ndc} = Z_{cc}/W_{cc}$$

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 [GLUT example](#) 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. [The previous question contains information on how to do this.](#)

Higher-level APIs, such as Fahrenheit 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 o
glStencilFunc (GL_ALWAYS, 0x1, 0x1);

/* Render a set of geometry corresponding to the area outside the view
/* The stencil buffer now has a single bit painted in the area outside

/* 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 [the previous question](#) 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  $2^{(\text{redBits}+\text{greenBits}+\text{blueBits})}$ .

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 th
   Given GLuint makeMask (GLint) returns a bit mask for the number of
*/

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++) {
    glColor3ui (indx & redMask << redShift,
               indx & greenMask << greenShift,
               indx & blueMask << blueShift);
    renderPrimitive (indx);
}

```

Also, make sure you disable any state that could alter the final color. [See the question 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 dramatic as moving the  $zNear$  clipping plane.

The [OpenGL Reference Manual](#) description for `glFrustum()` relates depth precision to the  $zNear$  and  $zFar$  clipping planes by saying that roughly  $\log_2(zFar/zNear)$  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 [the last question in this section](#) 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

$$\text{glFrustum}(l, r, b, t, n, f);$$

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:

$$\begin{aligned} z_{ndc} &= z_c / w_c = [ -z_e * (f+n)/(f-n) - w_e * 2*f*n/(f-n) ] / -z_e \\ &= (f+n)/(f-n) + (w_e / z_e) * 2*f*n/(f-n) \end{aligned}$$

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  $z_w$

$$\begin{aligned} z_e / w_e &= f*n/(f-n) / ((z_w / s) - 0.5 * (f+n)/(f-n) - 0.5) \\ &= f * n / ((z_w / s) * (f-n) - 0.5 * (f+n) - 0.5 * (f-n)) \end{aligned}$$

$$= f * n / ((z_w / s) * (f-n) - f) [*]$$

Now let's look at two points, the  $zNear$  clipping plane and the  $zFar$  clipping plane:

$$z_w = 0 \Rightarrow z_e / w_e = f * n / (-f) = -n$$

$$z_w = s \Rightarrow z_e / w_e = f * n / ((f-n) - f) = -f$$

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  $s-1$ :

$$z_w = 1 \Rightarrow z_e / w_e = f * n / ((1/s) * (f-n) - f)$$

$$z_w = s-1 \Rightarrow z_e / w_e = f * n / (((s-1)/s) * (f-n) - f)$$

Now let's plug in some numbers, for example,  $n = 0.01$ ,  $f = 1000$  and  $s = 65535$  (i.e., a 16-bit depth buffer)

$$z_w = 1 \Rightarrow z_e / w_e = -0.01000015$$

$$z_w = s-1 \Rightarrow z_e / w_e = -395.90054$$

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  $z_w$

$$d(z_e / w_e) / d z_w = -f * n * (f-n) * (1/s) / ((z_w / s) * (f-n) - f)^2$$

Now evaluate it at  $z_w = s$

$$\begin{aligned} d(z_e / w_e) / d z_w &= -f * (f-n) * (1/s) / n \\ &= -f * (f/n-1) / s [**] \end{aligned}$$

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 *Zfighting*.

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 [GLUT example](#), 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 [OpenGL Programming Guide](#) outlines one well-know method. The algorithm for drawing a polygon and its outline is as follows:

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(). [See more information on gluUnProject\(\) here.](#)

#### 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 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. [See the transparency section](#) 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 [OpenGL Programming Guide, Third Edition](#), 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 [screen door 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 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. Screenshot transparency is sometimes preferable to blending, because 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 programmatically 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 [OpenGL call logging utility](#). 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, [texture objects](#), 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.

# 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.

[You can find instructions on using texture mapped fonts here.](#) This site also contains image files of fonts that you can download and read into your program to use as font texture maps. Additional useful information on other aspects of using OpenGL can be found at this site as well.

[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 here.](#)

[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

[The NeHe web page](#) 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?

[The NeHe web page](#) 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 [the NeHe web page](#) for a tutorial on using geometric fonts. Look for the tutorial on outline fonts.

See [the Free Type library](#).

GLTT, formerly available from <http://www.moonlight3d.org/glтт/>, supports geometric TrueType fonts. [Click here to download GLTT](#) (~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 [18.020](#).

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 [question 18.080](#) 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 [light maps](#).

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 [question 18.050](#).

## 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 [OpenGL Programming Guide](#) 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,...)
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 light's position
Set the light position //glLightfv(GL_LIGHT_POSITION,...)
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 effects 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?

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 [OpenGL Programming Guide](#), 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 [download the Mesa source code distribution](#), 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 [GL\\_SELECTION render mode](#) 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. [Here's information on rendering each primitive in a unique color](#) and [information on using `glDrawPixels\(\)`](#).

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 `gluUnProject()` 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  $(\text{windowHeight}-y)$ .

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 [question 21.070](#) 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 texture memory, set the priority of each texture object with `glPrioritizeTextures()` to minimize texture memory thrashing.
- When your application renders the scene, bind each texture object before rendering the geometry 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. [See this example code](#) for how to do this on HP systems. Many OpenGL vendors have settled on an the [EXT\\_separate\\_specular\\_color extension](#).

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. [See this example code](#) 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 [texture maps that were stored in display lists](#).

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 [shared across 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 [display lists can](#). 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. `ARB_multitexture` is the preferred multitexturing extension.

The `ARB_multitexture` spec is included in the OpenGL 1.2.1 spec:  
<http://www.opengl.org/Documentation/Specs.html>.

An example is on [Michael Gold's web page](#).

#### **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.

[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:

```
glTexImage2D(GL_PROXY_TEXTURE_2D, level, internalFormat,
            width, height, border, format, type, NULL);
```

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:

```
GLint width;

glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0,
                        GL_TEXTURE_WIDTH, &width);

if (width==0) {
    /* Can't use that texture */
}
```

#### 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 [OpenGL on Silicon Graphics Systems](#). 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 sheer 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?

You usually do this by getting the system time, doing some rendering, and getting the system time again. The difference between the two time measurements tells you how long it took to render. You can do other quick calculations to determine frames per second, triangles per second, and vertices per second.

Calculating pixels per second is a little tougher. The easiest way to calculate it is to write a small benchmark program that renders primitives of a known pixel size.

Some benchmark software is free to download. GLUT 3.7 comes with a benchmark called `progs/bucciarelli/gltest` that measures OpenGL rendering performance. You can also visit the [Standard Performance Evaluation Corporation](#), which has many benchmarks you can download and 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 it's 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.

## 23 Extensions and Versions

### 23.010 Where can I find information on different OpenGL extensions?

The [OpenGL extension registry](#) is the central resource for all OpenGL extensions. Also, the [OpenGL.org web page](#) maintains a lot of information on different OpenGL 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:

```
#ifndef 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. [Check for the extension string](#) 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, [check for the extension string](#) 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:

```
#include <string.h>
...
const GLubyte *str;
int glPolyOffExtAvailable;

str = glGetString (GL_EXTENSIONS);
glPolyOffExtAvailable = (strstr((const char *)str, "GL_EXT_polygon_off
    != NULL);
```

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
.h file, or GL/glExt.h as shown here, which contains definitions fo
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.

### 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.

Linux uses the [OpenGL ABI](#).

### 23.090 Where can I find extension enumerants and function prototypes?

See the [OpenGL extension registry](#).

## 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 in fill mode with color set to the background color, and again in line mode. Use polygon offset so the lines over the polygons render correctly. [The polygon offset section](#) may be helpful to you.

Often you need to preserve a nonuniform background, such as a gradient fill or an image. In this case, do 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. [ModelMagic3D](#) is shareware and comes with source code. [GLScene](#) 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.

[Crossroads](#) can import many file formats and output the data as C/C++ compilable data that is suitable for use with vertex arrays.

This [file format information](#) covers a variety of different file formats.

[3DWinOGL](#) 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.

Download the [GLUT](#) 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: <http://www.helsinki.fi/~tksuoran/lw.html> and <http://www.cs.helsinki.fi/~tksuoran/glelite/>.

[3D Exploration](#) 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.?**

The easiest method is to use any of a number of image utilities that let you capture the screen or window, and save it as 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.

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](#).

The [gd library lets you create JPG and PNG files](#) from within your program.

[Imlib](#) 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 [section on shadows](#) or the [section on texture mapping](#) 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 [Opgl Programming Guide, Third Edition](#), 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 [evaluate ray intersections](#) (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 [library for collision detection called I COLLIDE](#) 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 executed 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 which 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.

There's a free utility called GLTrace2, which appears to contain similar functionality to ZAPdb. [More info on GLTrace2 can be found here.](#)

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?**

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.