



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.

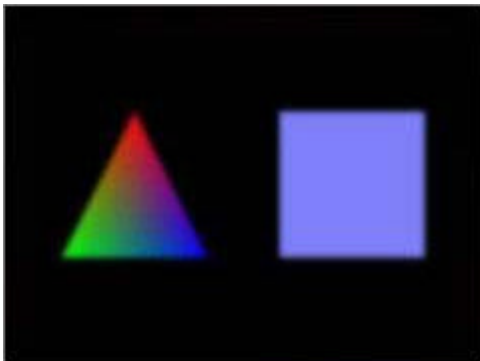
Your First Polygon:



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).

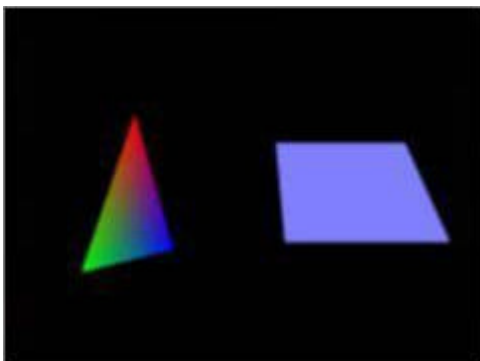
Colors:



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.

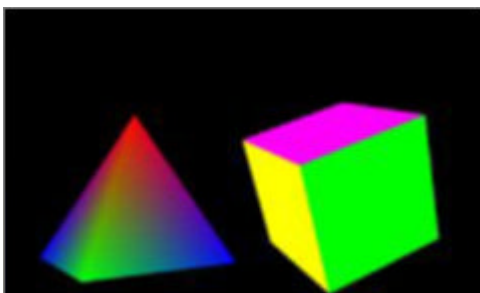
Rotation:



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:

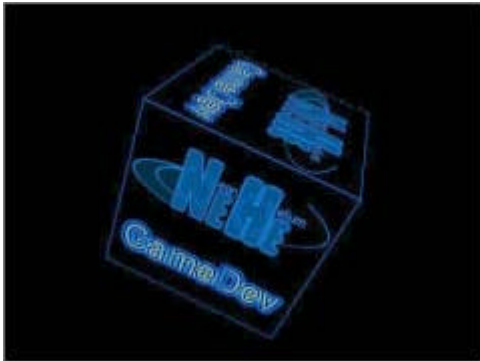


Now that we have setup, polygons, quads, colors and rotation figured out, it's time to build 3D objects. We'll build the objects using polygons and quads. This time we'll expand on the last tutorial, and turn the triangle into a colorful pyramid, and turn the square into a solid cube. The pyramid will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

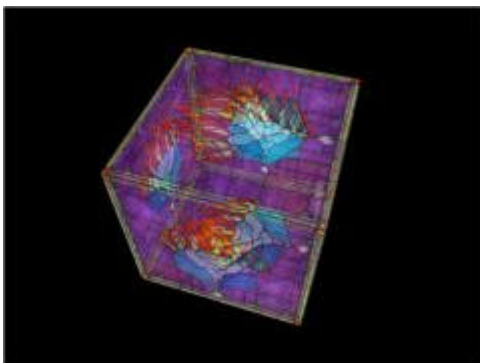
You'll find the code from lesson one is extremely valuable when it comes to developing a project quickly. The code in lesson one sets everything up for you, all you have to do is concentrate on programming the effect(s).

Texture Filters, Lighting & Keyboard Control:



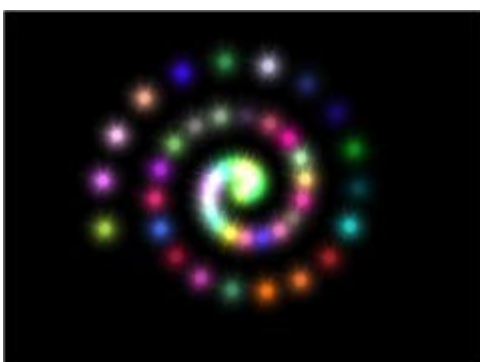
Ok, I hope you've been understanding everything up till now, because this is a huge tutorial. I'm going to attempt to teach you 2 new ways to filter your textures, simple lighting, keyboard control, and probably more :) If you don't feel confident with what you've learned up to this lesson, go back and 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 a lot of effort into the tutorial, so let him know what you think. Blending is not an easy topic to cover.

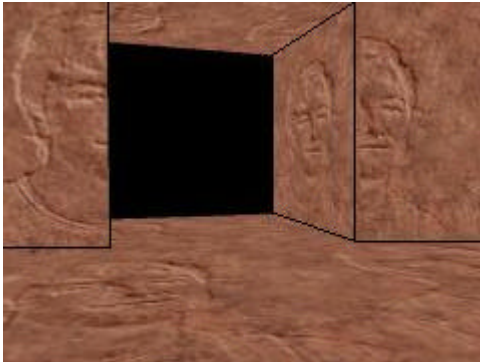
Moving Bitmaps In 3D Space:



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:**



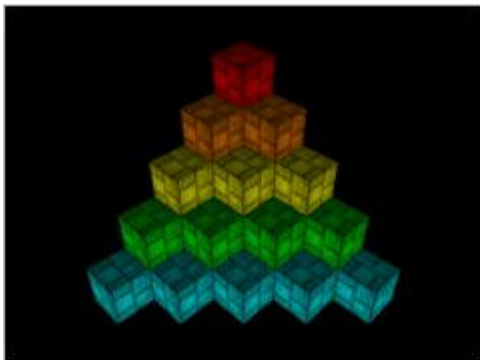
The tutorial you have all been waiting for! This tutorial was made by a fellow programmer named [Lionel Brits](#). In this lesson you will learn how to load a 3D world from a data file, and move through the 3D world. The code is made using lesson 1 code, however, the tutorial web page only explains the 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:**



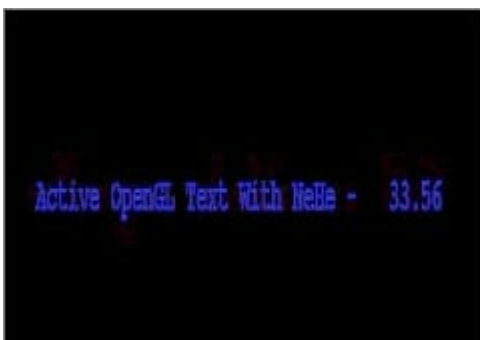
This tutorials brought to you by Bosco. The same guy that created the totally cool mini demo called worthless. He enjoyed everyones reaction to his demo, and decided to go one step further and explain how he does the cool effect at the end of his demo. This tutorial builds on the code from lesson 6.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

Display Lists:



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

Bitmap Fonts:



I think the question I get asked most often in email is "how can I display text on the screen using OpenGL?". You could always texture map text onto your screen. Of course you have very little control over the text, and unless you're good at blending, the text usually ends up mixing with the images on the screen. If you'd like an easy way to write the text you want anywhere you want on the screen in any color you want, using any of your computers built in fonts, then this tutorial is definitely for you. Bitmaps font's



are 2D scalable fonts, they can not be rotated. They always face forward.

Outline Fonts:



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

Texture Mapped Fonts:



Hopefully my last font tutorial {grin}. This time we learn a quick and fairly nice looking way to texture map fonts, and any other 3D object on your screen. By playing around with the code, you can create some pretty cool special effects, Everything from normal texture mapped object to sphere mapped objects. In case you don't know... Sphere mapping creates a metallic looking object that reflects anything from a pattern to a picture.

*** Cool Looking Fog:**



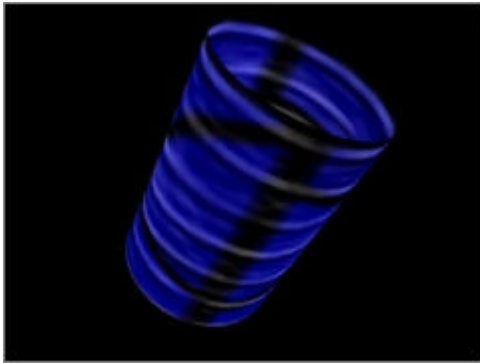
This tutorial was generously donated to the site by Chris Aliotta. It based on the code from lesson 7, that why you're seeing the famous crate again :) It's a pretty short tutorial aimed at teaching you the art of fog. You'll learn how to use 3 different fog filters, how to change the color of the fog, 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:**



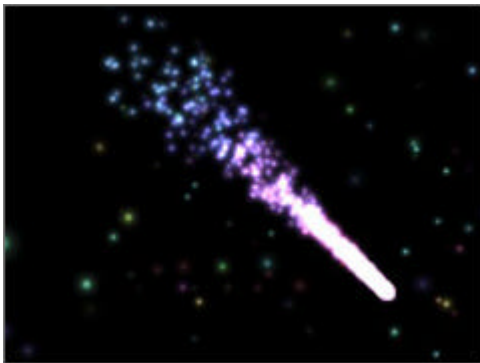
The original version of this tutorial was written by Giuseppe D'Agata. In this tutorial you will learn how to write any character or phrase you want to the screen using texture mapped quads. You will learn how to read one of 256 different characters from a 256x256 texture map, and finally I will show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



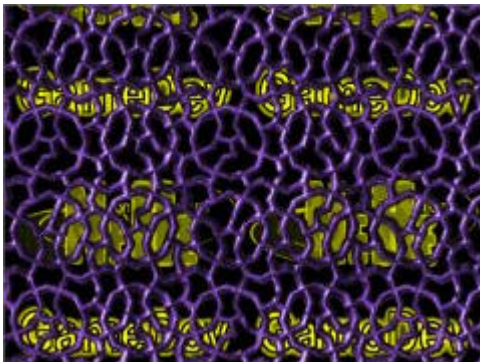
This tutorial was written by GB Schmick the wonderful site op over at [TipTup](#). It will introduce you to the wonderful world of quadratics. With quadratics you can easily create complex objects such as spheres, discs, cylinders and cones. These object can be created with just one line of code. With 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:



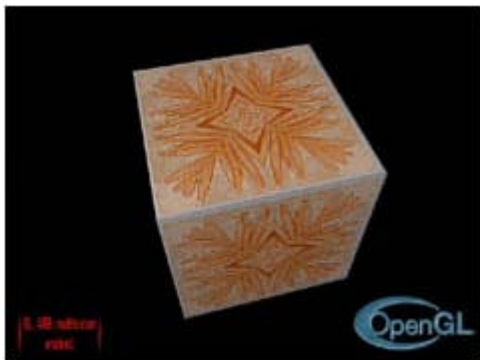
Up until now we've been blending our images onto the screen. Although this is effective, and it adds our image to the scene, a transparent object is not always pretty. Lets say you're making a game and you want solid text, or an odd shaped console to pop up. With regular blending this is not possible. 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 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!

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!

[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

[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 you gcc version.

The OpenGL library

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

```
> cd /usr/openwin/lib  
> ls libGL*
```

This should print:

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

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

But are the header files also there?

```
> cd /usr/openwin/include/GL  
> ls
```

This should print:

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

I have it. But what version is it?

This is a FAQ.

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

Helps you with questions dealing with OpenGL on Sun platforms.

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

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

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

Helps you. Make sure to get the 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 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 Support
{
```

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:                                     // Watch :
{
    if (!HIWORD(wParam))
    {
        active=TRUE;
    }
    else
    {
        active=FALSE;
    }

    return 0;                                         // Return
}

```

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

```

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

```

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

```

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!](#)

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 Herborth](#))
- * DOWNLOAD [Java](#) Code For This Lesson. (Conversion by [Darren Hodges](#))

[Back To NeHe Productions!](#)

Lesson 3

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

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

```
int DrawGLScene(GLvoid)                                     // Here's Where We
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);    // Clear The Screen And The
    glLoadIdentity();                                       // Reset The Current 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 dont have to move 1.5 from left to center then 1.5 to the right (total of 3.0) we only have to move from center to the right which is just 1.5 units.

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

```
glLoadIdentity(); // Reset The Current Modelv
glTranslatef(1.5f,0.0f,-6.0f); // Move Right 1.5 U
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 B
glBegin(GL_QUADS); // Start Drawing A Quad
    glVertex3f(-1.0f, 1.0f, 0.0f); // Top Left Of The
    glVertex3f( 1.0f, 1.0f, 0.0f); // Top Right Of The
    glVertex3f( 1.0f,-1.0f, 0.0f); // Bottom Right Of
    glVertex3f(-1.0f,-1.0f, 0.0f); // Bottom Left Of T
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 Rot
rquad-=0.15f; // Decrease The Rot
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 The
glVertex3f( 1.0f,-1.0f, 1.0f);     // Bottom Right Of The

```

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

```

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

```

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 The
glVertex3f( 1.0f,-1.0f,-1.0f);     // Bottom Right Of The
glEnd();                             // Done Drawing The Quad

rtri+=0.2f;                           // Increase The Rotation
rquad-=0.15f;                          // Decrease The Rotation
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; // Private
HGLRC        hRC=NULL; // Permanent
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 '
```

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 interested 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.

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

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_s, A_s, A_s, A_s) (AKA source alpha) for S and (1, 1, 1, 1) - (A_s, A_s, A_s, A_s) (AKA one minus src alpha) for D. This will yield a blending equation that looks like this:

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

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
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.

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 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

    // 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
{
    glClear(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 1
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 Drawing
        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 Tri

}
return TRUE; // Jump Back
}

```

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

```

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

```

glTexCoord2f( float_x, float_y); // First Texture Coordinate
glVertex3f( points[x][y][0], points[x][y][1], points[x][y][2] );

glTexCoord2f( float_x, float_yb ); // Second Texture Coordinate
glVertex3f( points[x][y+1][0], points[x][y+1][1], points[x][y+1][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)

- * [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/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 State
// Window Active Flag Set To TRUE
// 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 Cubes
}
if (keys[VK_UP])                        // Up Arrow Being Pressed?
{
    xrot-=0.2f;                          // If So Tilt Cubes
}
if (keys[VK_DOWN])                     // Down Arrow Being Pressed:
{
    xrot+=0.2f;                          // If So Tilt Cubes
}

```

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 Client Array
```

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 Corrections

    BuildFont();                                          // Build The Font

    return TRUE;                                          // Initialization Was Successful
}

```

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+cnt:

```

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

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

```

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

```

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

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

```

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 L
    glEnable(GL_LIGHTING);                              // Enable Lighting
    glEnable(GL_COLOR_MATERIAL);                        // Enable Coloring

    BuildFont();                                        // Build The Font

    return TRUE;                                        // Initialization I
}

```

Now for the drawing code. We start off by clearing the screen and the depth buffer. We call `glLoadIdentity()` to reset everything. Then we translate 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(co:
```

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 Keyboa
bool active=TRUE; // Window Active Flag Set To
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen
```

We're going to add one new integer variable here called **texture[]**. It will be used to store our texture. The last three lines were in tutorial 14 and have not changed in this tutorial.

```
GLuint texture[1]; // One Texture Map ( NEW )
GLuint base; // Base Display List For The

GLfloat rot; // Used To Rotate The Text

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

The following section of code has some minor changes. In this tutorial I'm going to use the wingdings font to display a skull and crossbones type object. If you want to display text instead, you can leave the code the same as it was in lesson 14, or change to a font of your own.

A few of you were wondering how to use the wingdings font, which is another reason I'm not using a standard font. Wingdings is a SYMBOL font, and requires a few changes to make it work. It's not as easy as telling Windows to use the wingdings font. If you change the font name to wingdings, you'll notice that the font doesn't get selected. You have to tell Windows that the font is a symbol font and not a standard character font. More on this later.

```

GLvoid BuildFont(GLvoid)                                // Build Our Bitmap Font
{
    GLYPHMETRICSFLOAT gmf[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 ( Mod:

```

The remaining lines of code have not changed.

```
SelectObject(hDC, font); // Selects The Font We Creat

wglUseFontOutlines(      hDC, // Select The Curre
                        0, // Starting Charact
                        255, // Number Of Displa
                        base, // Starting Display
```

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 f
```

The next three lines of code are still the same.

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

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 File
    {
        return NULL; // If Not Return NU

    }

    File=fopen(Filename,"r"); // Check To See If The File

    if (File) // Does The File Exist?
    {
        fclose(File); // Close The Handle
        return auxDIBImageLoad(Filename); // Load The Bitmap And Retur
    }

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

```

}

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

```

```

        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. 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")) && // Load T
    (TextureImage[1]=LoadBMP("Data/Bumps.bmp"))) // Load T

```

```

    {
        Status=TRUE;
    }

```

Another important line to double check. I can't begin to tell you how many emails I've received from people asking *"why am I only seeing one texture, or why are my textures all white!?!"*. Usually this line is the problem. If you were to replace the 2 with a 1, only one texture would be created and the second texture would appear all white. If you replaced the 2 with a 3 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]);
            }
        }
        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);
glBegin(GL_QUADS);
    glTexCoord2d(0.0f,0.0f);
    glVertex2f(-1.0f, 1.0f);
    glTexCoord2d(1.0f,0.0f);
    glVertex2f( 1.0f, 1.0f);
    glTexCoord2d(1.0f,1.0f);
    glVertex2f( 1.0f,-1.0f);
    glTexCoord2d(0.0f,1.0f);
    glVertex2f(-1.0f,-1.0f);
glEnd();

```

// Bright
// Draw O
// First '
// First '
// Second
// Second
// Third '
// Third '
// Fourth
// Fourth
// 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);
glBegin(GL_QUADS);
    glTexCoord2d(0.0f,0.0f);
    glVertex2f(-1.0f, 1.0f);
    glTexCoord2d(1.0f,0.0f);
    glVertex2f( 1.0f, 1.0f);
    glTexCoord2d(1.0f,1.0f);
    glVertex2f( 1.0f,-1.0f);
    glTexCoord2d(0.0f,1.0f);
    glVertex2f(-1.0f,-1.0f);
glEnd();

```

// Rotate
// Draw O
// First '
// First '
// Second
// Second
// Third '
// Third '
// Fourth
// Fourth
// 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 slow 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 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, Destination One
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Corrections
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST); // Really Nice Point Smoothing
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

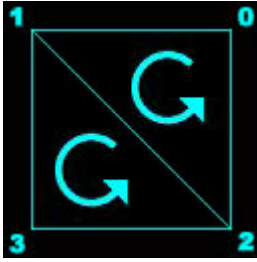
```

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].xg+=1.5f;
```

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].xg-=1.5f;
```

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, // Instance
    HINSTANCE PrevInstance, // Previous Instance
    LPSTR lpCmdLine, // Command Line
    int nCmdShow) // Window Show State
{
    MSG msg; // Window Message
    BOOL done=FALSE; // Bool Variable

    // Ask The User Which Screen Mode They Prefer
    if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start FullScreen",
        MB_YESNO) == IDYES)
    {
        fullscreen=TRUE; // Fullscreen Mode
    }
    else
    {
        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 Increase
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 Increase
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 Color
```

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 Window
    fullscreen=!fullscreen; // Toggle Fullscreen
    // Recreate Our OpenGL Window
    if (!CreateGLWindow("NeHe's Particle Tutorial")
    {
        return 0; // Quit If Window Was Not Created
    }
}
}
}
// 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;
}
```

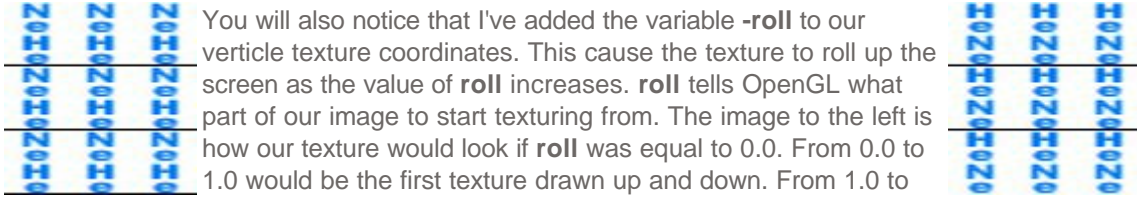
Now for the fun stuff. Our drawing code! We start off the same as usual. We clear the background color and the depth buffer. Then we reset the modelview matrix, and translate into the screen 2 units so that we can see our scene.

```
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 texture would be drawn from -0.5 to 0.5 (notice that because we started drawing halfway through the texture that the 'N' and 'e' have been cut off). The second texture would be from 0.5 to 1.5, and the third texture would be from 1.5 to 2.5. Again notice that only the 'N' and 'e' have been drawn at the bottom. We never quite made it to 3.0 (the bottom of a complete texture) so the 'H' and 'e' were not drawn. Rolling textures can be used to create great effects such as moving clouds. Words spinning around an object, etc.



If you don't understand what I mean about rolling textures, let me know. If you have a better way to explain let me know. It's easy to understand how rolling textures work once you've used them, but trying to explain it in words isn't very easy.

One last explanation to hopefully clear things up. Imagine you had an endless amount of marbles up and down, left and right. Every marble was 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 of FALSE. If the program detects a performance counter, the variable **performance_timer** variable is set to TRUE, and all timing is done using the performance counter (alot more accurate than the multimedia timer). If a performance counter is not found, **performance_timer** is set to FALSE and the multimedia timer is used for timing.

The last 2 variables are 64 bit integer variables that hold the start time of the performance counter and the amount of time that has elapsed since the performance counter was started.

The name of this structure is "timer" as you can see at the bottom of the structure. If we want to know the timer frequency we can now check **timer.frequency**. Nice!

```
struct
{
    __int64      frequency;                // Timer :
    float       resolution;              // Timer :
    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 Performance 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)

{
    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 P
        {
            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 wont play again.

After we've played the sound we give the hourglass a random value on the x-axis. We add one to the random value so that the hourglass doesn't appear at the players starting position at the top left of the grid. We also give the hourglass a random value on the y-axis. We set **hourglass.fx** to 1 this makes the hourglass appear on the screen at it's new location. We also set **hourglass.fy** back to zero so it can start counting again.

This causes the hourglass to appear on the screen after a fixed amount of time.

```
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_l** is light diffuse color
- **D_m** 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_d**
- (**F_d+m**) approximates (**L*N**) per pixel

Approximate Derivative

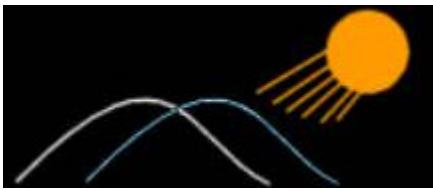
Embossing Approximates Derivative

- Lookup height **H₀** at point (**s,t**)
- Lookup height **H₁** at point slightly perturbed toward light source (**s+ds,t+dt**)
- Subtract original height **H₀** from perturbed height **H₁**
- Difference represents instantaneous slope **m=H₁-H₀**

Compute The Bump



1) Original bump (**H₀**).



2) Original bump (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.

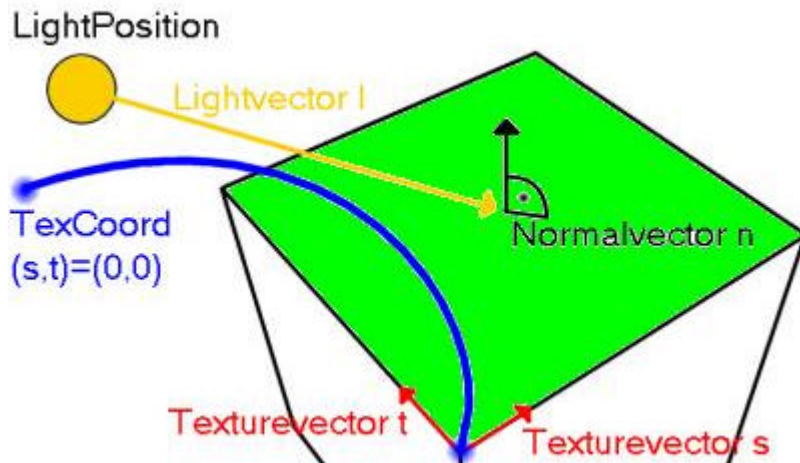
- We calculate the **current vertex c** on our surface (simply by looking it up in **data**).
- Then we'll calculate a normal **n** with length 1 (We usually know **n** for each face of a cube!). This is important, since we can save computing time by requesting normalized vectors. Calculate the **light 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 '
    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:

- 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!](#)