Neon Helium Productions!

OpenGL Tutorial

Contents

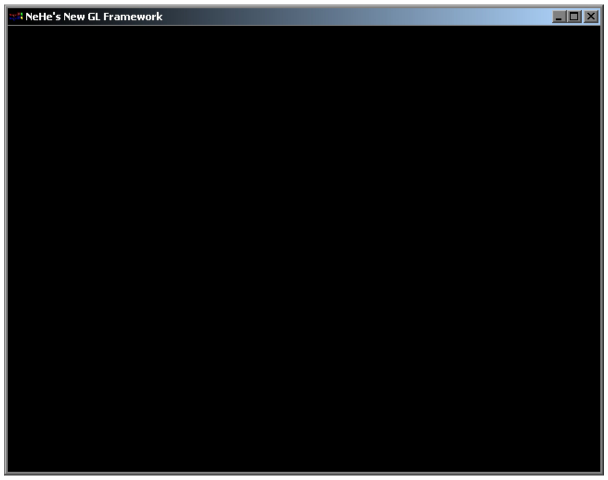
Preface

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

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

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

Lesson 01
Setting Up An OpenGL Window



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> // Header File For Windows
#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
```

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.

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

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]; // 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
```

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); // Declaration For WndProc
```

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 And Initialize The GL Window
{
if (height==0) // Prevent A Divide By Zero By
{
height=1; // Making Height Equal One
}

glViewport(0, 0, width, height); // Reset The Current Viewport
```

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); // 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,100.0f);

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

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

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
```

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); // Enables Smooth Shading
```

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

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

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Black Background
```

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); // Depth Buffer Setup
glEnable(GL_DEPTH_TEST); // Enables Depth Testing
glDepthFunc(GL_LEQUAL); // The Type Of Depth Test To Do
```

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 Nice Perspective Calculations
```

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; // Initialization Went OK
}
```

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 wont draw anything yet.

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

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The Current Modelview Matrix
return TRUE; // Everything Went OK
}
```

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) // Properly Kill The Window
{
```

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) // Are We In Fullscreen Mode?
{
```

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); // If So Switch Back To The Desktop
ShowCursor(TRUE); // Show Mouse Pointer
}
```

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 Have A Rendering Context?
{
```

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)) // Are We Able To Release The DC And RC Contexts?
{
```

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",MB_OK | MB_ICONINFORMATION);
}
```

Next we try to delete the Rendering Context. If we were unsuccessful an error message will pop up.

```
if (!wglDeleteContext(hRC)) // Are We Able To Delete The RC?
{
```

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 ERROR",MB_OK |
MB_ICONINFORMATION);
}
hRC=NULL; // Set RC To NULL
}
```

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 Able To Release The DC
{
MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hDC=NULL; // Set DC To NULL
}
```

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 Able To Destroy The Window?
{
MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hWnd=NULL; // Set hWnd To NULL
}
```

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

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

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; // Holds The Results After Searching For A Match
```

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; // Windows Class Structure
```

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; // Window Extended Style
DWORD dwStyle; // Window Style
```

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 Rectangle Upper Left / Lower Right Values
WindowRect.left=(long)0; // Set Left Value To 0
WindowRect.right=(long)width; // Set Right Value To Requested Width
WindowRect.top=(long)0; // Set Top Value To 0
WindowRect.bottom=(long)height; // Set Bottom Value To Requested 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 The Global Fullscreen Flag
```

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 For Our Window
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // Redraw On Move, And Own DC For Window
wc.lpfnWndProc = (WNDPROC) WndProc; // WndProc Handles Messages
wc.cbClsExtra = 0; // No Extra Window Data
wc.cbWndExtra = 0; // No Extra Window Data
wc.hInstance = hInstance; // Set The Instance
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Load The Default Icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Load The Arrow Pointer
wc.hbrBackground = NULL; // No Background Required For GL
wc.lpszMenuName = NULL; // We Don't Want A Menu
wc.lpszClassName = "OpenGL"; // Set The Class Name
```

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 To Register The Window Class
{
MessageBox(NULL,"Failed To Register The Window Class.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Exit And 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) // Attempt Fullscreen Mode?
{
```

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 Mode
memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's Cleared
dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode Structure
dmScreenSettings.dmPelsWidth = width; // Selected Screen Width
dmScreenSettings.dmPelsHeight = height; // Selected Screen Height
dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pixel
dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;
```

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 Rid Of Start Bar.
if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
{
```

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 Window.
if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use
Windowed Mode Instead?","NeHe GL",MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
{
```

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

```
fullscreen=FALSE; // Select Windowed Mode (Fullscreen=FALSE)
}
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 Is Closing.
MessageBox(NULL,"Program Will Now Close.","ERROR",MB_OK|MB_ICONSTOP);
return FALSE; // Exit And 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) // Are We Still In Fullscreen Mode?
{
```

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 Extended Style
dwStyle=WS_POPUP; // Windows Style
ShowCursor(FALSE); // Hide Mouse Pointer
}
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 Extended Style
dwStyle=WS_OVERLAPPEDWINDOW; // Windows Style
}
```

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 Window To True Requested
Size
```

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, // Extended Style For The Window
"OpenGL", // Class Name
title, // Window Title
WS_CLIPSIBLINGS | // Required Window Style
WS_CLIPCHILDREN | // Required Window Style
dwStyle, // Selected Window Style
0, 0, // Window Position
WindowRect.right-WindowRect.left, // Calculate Adjusted Window Width
WindowRect.bottom-WindowRect.top, // Calculate Adjusted Window Height
NULL, // No Parent Window
NULL, // No Menu
hInstance, // Instance
NULL))) // Don't Pass Anything To WM_CREATE
```

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(); // Reset The Display
MessageBox(NULL,"Window Creation Error.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // 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 Tells Windows How We Want Things To Be
{
sizeof(PIXELFORMATDESCRIPTOR), // Size Of This Pixel Format Descriptor
1, // Version Number
PFD_DRAW_TO_WINDOW | // Format Must Support Window
PFD_SUPPORT_OPENGL | // Format Must Support OpenGL
PFD_DOUBLEBUFFER, // Must Support Double Buffering
PFD_TYPE_RGBA, // Request An RGBA Format
bits, // Select Our Color Depth
0, 0, 0, 0, 0, 0, // Color Bits Ignored
0, // No Alpha Buffer
0, // Shift Bit Ignored
0, // No Accumulation Buffer
0, 0, 0, 0, // Accumulation Bits Ignored
16, // 16Bit Z-Buffer (Depth Buffer)
0, // No Stencil Buffer
0, // No Auxiliary Buffer
PFD_MAIN_PLANE, // Main Drawing Layer
0, // Reserved
0, 0, 0 // Layer Masks Ignored
};
```

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))) // Did We Get A Device Context?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Create A GL Device Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // 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))) // Did Windows Find A Matching Pixel Format?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Find A Suitable PixelFormat.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // 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 Able To Set The Pixel Format?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Set The PixelFormat.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // 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 Able To Get A Rendering Context?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Create A GL Rendering Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);

return FALSE; // 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)) // Try To Activate The Rendering Context
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Activate The GL Rendering Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // 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 The Window
SetForegroundWindow(hWnd); // Slightly Higher Priority
SetFocus(hWnd); // Sets Keyboard Focus To The Window
ReSizeGLScene(width, height); // Set Up Our Perspective GL Screen
```

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

```
if (!InitGL()) // Initialize Our Newly Created GL Window
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Initialization Failed.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // 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; // Success
}
```

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

```
LRESULT CALLBACK WndProc( HWND hWnd, // Handle For This Window
UINT uMsg, // Message For This Window
WPARAM wParam, // Additional Message Information
LPARAM lParam) // Additional Message Information
{
```

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) // Check For Windows Messages
{
```

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 For Window Activate Message
{
if (!HIWORD(wParam)) // Check Minimization State
{
active=TRUE; // Program Is Active
}
else
{
active=FALSE; // Program Is No Longer Active
}
return 0; // Return To The Message Loop
}
```

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: // Intercept System Commands
{
switch (wParam) // Check System Calls
{
case SC_SCREENSAVE: // Screensaver Trying To Start?
case SC_MONITORPOWER: // Monitor Trying To Enter Powersave?
return 0; // Prevent From Happening
}
break; // Exit
}
```

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: // Did We Receive A Close Message?
{
PostQuitMessage(0); // Send A Quit Message
return 0; // Jump Back
}
```

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 Key Being Held Down?
{
keys[wParam] = TRUE; // If So, Mark It As TRUE
return 0; // Jump Back
}
```

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: // Has A Key Been Released?
{
keys[wParam] = FALSE; // If So, Mark It As FALSE
return 0; // Jump Back
}
```

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: // Resize The OpenGL Window
{
ReSizeGLScene(LOWORD(lParam),HIWORD(lParam)); // LoWord=Width, HiWord=Height
return 0; // Jump Back
}
}
```

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

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; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop
```

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?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}
```

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 If Window Was Not Created
}
```

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

```
while(!done) // Loop That Runs Until done=TRUE
{
```

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 There A Message Waiting?
{
```

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 We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
```

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); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
```

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. If 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 DrawGLScene()
if (active) // Program Active?
{
if (keys[VK_ESCAPE]) // Was ESC Pressed?
{
done=TRUE; // ESC Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
```

If the program is active and esc was not pressed we render the scene and 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.

```
DrawGLScene(); // Draw The Scene
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
}
}
```

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 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's OpenGL Framework",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
}
}
```

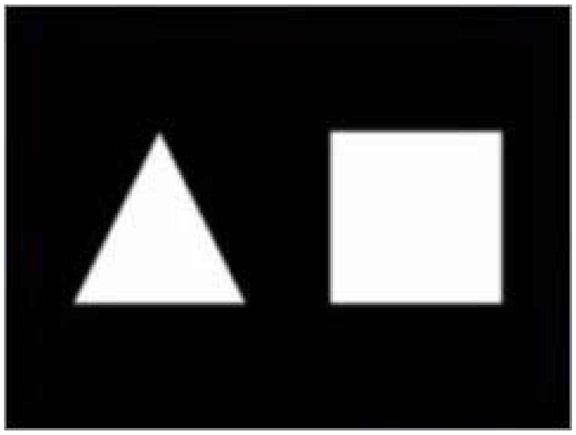
If the done variable is no longer FALSE, the program quits. We kill the OpenGL window properly so that everything is freed up, and we exit the program.

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

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**)

Lesson 02
Your First Polygon



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 Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
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 Left 1.5 Units And Into The Screen 6.0
```

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 render objects as 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. This applies to quads as well. If you know you're drawing all quads, you can include the second group of 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); // Drawing Using Triangles
glVertex3f( 0.0f, 1.0f, 0.0f); // Top
glVertex3f(-1.0f,-1.0f, 0.0f); // Bottom Left
glVertex3f( 1.0f,-1.0f, 0.0f); // Bottom Right
glEnd(); // Finished Drawing The Triangle
```

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 Right 3 Units
```

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 (clockwise). By drawing in a clockwise order, the square will be drawn as a back face. Meaning the side of the quad we see is actually the back. Objects drawn in a counter clockwise order will be facing us. Not important at the moment, but later on you will need to know this.

```
glBegin(GL_QUADS); // Draw A Quad
glVertex3f(-1.0f, 1.0f, 0.0f); // Top Left
glVertex3f( 1.0f, 1.0f, 0.0f); // Top Right
glVertex3f( 1.0f,-1.0f, 0.0f); // Bottom Right
glVertex3f(-1.0f,-1.0f, 0.0f); // Bottom Left
glEnd(); // Done Drawing The 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 Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window ( Modified )
if (!CreateGLWindow("NeHe's First Polygon Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
```

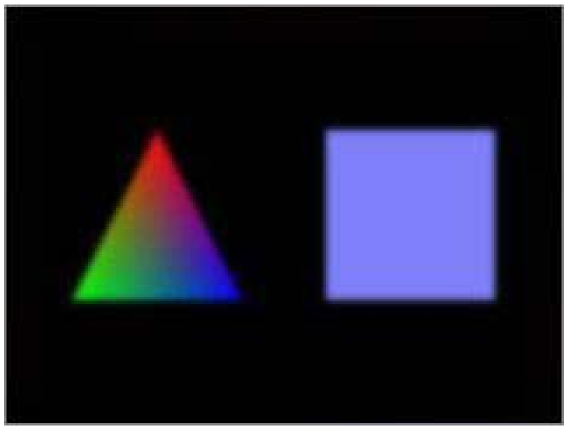
Markus Knauer Adds: In the book ("OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1", J. Neider, T. Davis, M. Woo, Addison-Wesley, 1993) the following paragraph will clearly explain what NeHe means when he refers to movement by units in OpenGL:

"[I mentioned] inches and millimeters - do these really have anything to do with OpenGL? The answer is, in a word, no. The projection and other transformations are inheritly unitless. If you want to think of the near and far clipping planes as located at 1.0 and 20.0 meters, inches, kilometers, or leagues, it's up to you. The only rule is that you have to use consistent unit of measurement."

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**)

Lesson 03
Adding Color



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 Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The Current Modelview Matrix

glTranslatef(-1.5f,0.0f,-6.0f); // Left 1.5 Then Into Screen Six Units

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

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 From Center (Top Point)
```

We've placed the first vertex on the screen, setting it's color to red. Now before we set 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 One Unit (Bottom Left)
```

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 One Unit (Bottom Right)
glEnd(); // Done Drawing A Triangle

glTranslatef(3.0f,0.0f,0.0f); // From Right Point Move 3 Units Right
```

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 solid blue square. Again, the square (quad) is drawn in a clockwise order meaning we start off looking at the back of the quad.

```
glColor3f(0.5f,0.5f,1.0f); // Set The Color To Blue One Time Only
glBegin(GL_QUADS); // Start Drawing Quads
glVertex3f(-1.0f, 1.0f, 0.0f); // Left And Up 1 Unit (Top Left)
glVertex3f( 1.0f, 1.0f, 0.0f); // Right And Up 1 Unit (Top Right)
glVertex3f( 1.0f,-1.0f, 0.0f); // Right And Down One Unit (Bottom Right)
glVertex3f(-1.0f,-1.0f, 0.0f); // Left And Down One Unit (Bottom Left)
glEnd(); // Done Drawing A Quad
return TRUE; // Keep Going
}
```

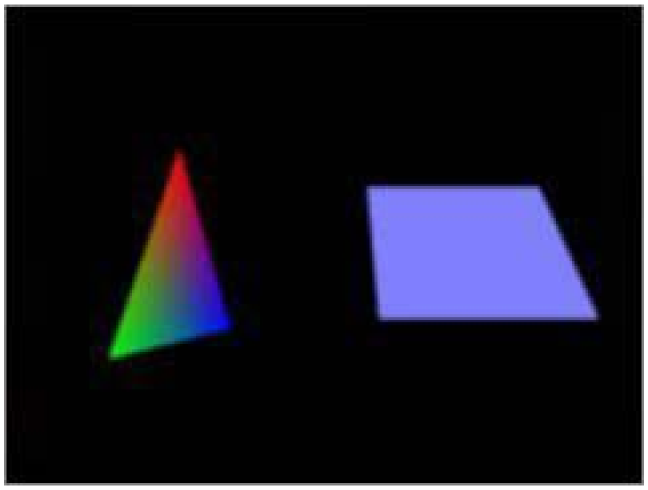
Finally change the code to toggle window / fullscreen mode so that the title at the top of the window is proper.

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

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**)

Lesson 04
Rotation



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 at the top of our program, underneath the other variables. You will notice two new lines after 'bool fullscreen=TRUE;'. 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 will find that floating point numbers are essential to OpenGL programming. The new variables are called rtri which will rotate the triangle and rquad which will rotate the quad.

```
#include <windows.h> // Header File For Windows
#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
bool fullscreen=TRUE; // Fullscreen Flag Set To TRUE By Default

GLfloat rtri; // Angle For The Triangle ( NEW )
GLfloat rquad; // Angle For The Quad ( NEW )
```

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 Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
glTranslatef(-1.5f,0.0f,-6.0f); // Move Into The Screen And Left
```

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

(-1,0,0) describes a vector that travels in a direction of 1 unit along the x axis, but this time towards the left.

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

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

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

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

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

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

It's important to note that rotations are done in degrees. If rtri had a value of 10, we would be rotating 10 degrees on the y-axis.

```
glRotatef(rtri,0.0f,1.0f,0.0f); // Rotate The Triangle On The Y axis ( NEW )
```

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

```
glBegin(GL_TRIANGLES); // Start Drawing A Triangle
glColor3f(1.0f,0.0f,0.0f); // Set Top Point Of Triangle To Red
glVertex3f( 0.0f, 1.0f, 0.0f); // First Point Of The Triangle
glColor3f(0.0f,1.0f,0.0f); // Set Left Point Of Triangle To Green
glVertex3f(-1.0f,-1.0f, 0.0f); // Second Point Of The Triangle
glColor3f(0.0f,0.0f,1.0f); // Set Right Point Of Triangle To Blue
glVertex3f( 1.0f,-1.0f, 0.0f); // Third Point Of The 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 Modelview Matrix
glTranslatef(1.5f,0.0f,-6.0f); // Move Right 1.5 Units And Into The Screen 6.0
glRotatef(rquad,1.0f,0.0f,0.0f); // Rotate The Quad On The X axis ( NEW )
```

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

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

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

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

```
rtri+=0.2f; // Increase The Rotation Variable For The Triangle ( NEW )
rquad-=0.15f; // Decrease The Rotation Variable For The Quad ( NEW )
return TRUE; // Keep Going
}
```

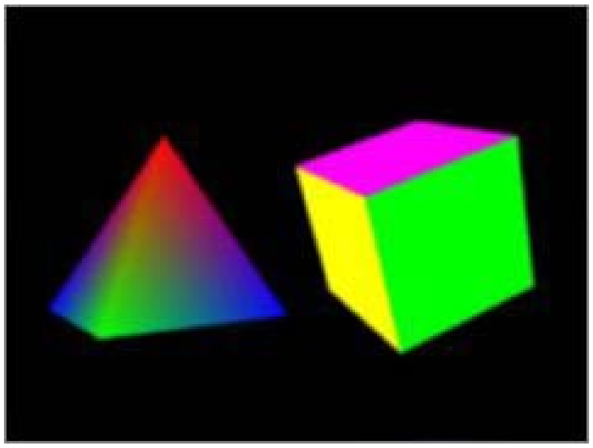
Finally change the code to toggle window / fullscreen mode so that the title at the top of the window is proper.

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

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**)

Lesson 05
3D Shapes



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 Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
glTranslatef(-1.5f,0.0f,-6.0f); // Move Left And Into The Screen

glRotatef(rtri,0.0f,1.0f,0.0f); // Rotate The Pyramid On It's Y Axis

glBegin(GL_TRIANGLES); // Start Drawing The Pyramid
```

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

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

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

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

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

glBegin(GL_QUADS); // Start Drawing The Cube
```

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

```
glColor3f(0.0f,1.0f,0.0f); // Set The Color To Green
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 Quad (Front)
glVertex3f(-1.0f, 1.0f, 1.0f); // Top Left Of The Quad (Front)
glVertex3f(-1.0f,-1.0f, 1.0f); // Bottom Left Of The Quad (Front)
glVertex3f( 1.0f,-1.0f, 1.0f); // Bottom Right Of The Quad (Front)
```

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); // Bottom Left Of The Quad (Back)
glVertex3f(-1.0f,-1.0f,-1.0f); // Bottom Right Of The Quad (Back)
glVertex3f(-1.0f, 1.0f,-1.0f); // Top Right Of The Quad (Back)
glVertex3f( 1.0f, 1.0f,-1.0f); // Top Left Of The Quad (Back)
```

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

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.

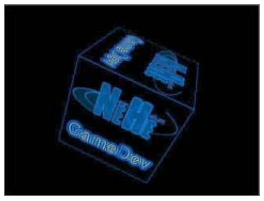
```
glColor3f(1.0f,0.0f,1.0f); // Set The Color To Violet
glVertex3f( 1.0f, 1.0f,-1.0f); // Top Right Of The Quad (Right)
glVertex3f( 1.0f, 1.0f, 1.0f); // Top Left Of The Quad (Right)
glVertex3f( 1.0f,-1.0f, 1.0f); // Bottom Left Of The Quad (Right)
glVertex3f( 1.0f,-1.0f,-1.0f); // Bottom Right Of The Quad (Right)
glEnd(); // Done Drawing The Quad

rtri+=0.2f; // Increase The Rotation Variable For The Triangle
rquad-=0.15f; // Decrease The Rotation Variable For The Quad
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**)

Lesson 06
Texture Mapping



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> // Header File For Windows
#include <stdio.h> // Header File For Standard Input/Output ( NEW )
#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
bool fullscreen=TRUE; // Fullscreen Flag

GLfloat xrot; // X Rotation ( NEW )
GLfloat yrot; // Y Rotation ( NEW )
GLfloat zrot; // Z Rotation ( NEW )

GLuint texture[1]; // Storage For One Texture ( NEW )

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

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 A Bitmap Image
{
FILE *File=NULL; // File Handle
```

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) // Make Sure A Filename Was Given
{
return NULL; // If Not 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 To See If The File Exists
```

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 The File Exist?
{
fclose(File); // Close The Handle
return auxDIBImageLoad(Filename); // Load The Bitmap And Return A Pointer
}
```

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

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

```
int LoadGLTextures() // Load Bitmaps And Convert To Textures
{
```

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 Indicator
```

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 Storage Space For The Texture
```

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

```
memset(TextureImage,0,sizeof(void *)*1); // Set The Pointer To NULL
```

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; // Set The Status To 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 generate one texture name (increase the number if you load more than one 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's not. 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 to bind the named texture texture[0] to a texture target. 2D textures have both height (on the Y axes) and width (on the X axes). The main function of glBindTexture is to assign a texture name to texture data. In this case we're telling OpenGL there is memory available at &texture[0]. When we create the texture, it will be stored in the memory that &texture[0] references.

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

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

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 Filtering
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR); // Linear Filtering
}
```

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

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

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; // Return The 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 Setup For OpenGL Goes Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine ( NEW )
{
return FALSE; // If Texture Didn't Load Return FALSE ( NEW )
}

glEnable(GL_TEXTURE_2D); // Enable Texture Mapping ( NEW )
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 Calculations
return TRUE; // Initialization Went OK
}
```

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 black. The depth buffer will also be cleared. The view will then be reset with glLoadIdentity().

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
glLoadIdentity(); // Reset The Current Matrix
glTranslatef(0.0f,0.0f,-5.0f); // Move Into The Screen 5 Units
```

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); // Rotate On The X Axis
glRotatef(yrot,0.0f,1.0f,0.0f); // Rotate On The Y Axis
glRotatef(zrot,0.0f,0.0f,1.0f); // Rotate On The Z Axis
```

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 Our Texture
```

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 Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Left Of The Texture and Quad
// Back Face
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Bottom Left Of The Texture and
Quad
// Top Face
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Bottom Right Of The Texture and Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
// Bottom Face
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
// Right face
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Left Of The Texture and Quad
// Left Face
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom Left Of The Texture and
Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
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; // X Axis Rotation
yrot+=0.2f; // Y Axis Rotation
zrot+=0.4f; // Z Axis Rotation
return true; // Keep Going
}
```

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

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

Jeff Molofee (**NeHe**)

Lesson 07 Texture Filters, Lightning & Keyboard Control

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> // 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

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
bool fullscreen=TRUE; // Fullscreen Flag
```

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; // Lighting ON / OFF
BOOL lp; // L Pressed?
BOOL fp; // F Pressed?
```

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; // X Rotation
GLfloat yrot; // Y Rotation
GLfloat xspeed; // X Rotation Speed
GLfloat yspeed; // Y Rotation Speed
GLfloat z=-5.0f; // Depth Into The Screen
```

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 Light Values ( NEW )
```

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 Light Values ( NEW )
```

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 Position ( NEW )
```

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; // Which Filter To Use
GLuint texture[3]; // Storage for 3 textures
```

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

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 A Bitmap Image
{
FILE *File=NULL; // File Handle

if (!Filename) // Make Sure A Filename Was Given
{
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 Pointer
}
return NULL; // If Load Failed 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() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator

AUX_RGBImageRec *TextureImage[1]; // Create Storage Space For The Texture

memset(TextureImage,0,sizeof(void *)*1); // Set The Pointer To NULL
```

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; // Set The Status To 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]); // Create Three Textures
```

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

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

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

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

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

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

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

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

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[0]->sizeY, GL_RGB,
GL_UNSIGNED_BYTE, TextureImage[0]->data); ( NEW )
}
```

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

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

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; // Return The 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 Setup For OpenGL Goes Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load Return FALSE
}

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
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 Calculations
```

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); // Setup The Ambient Light
```

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); // Setup The Diffuse Light
```

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); // Position The Light
```

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

```
glEnable(GL_LIGHT1); // Enable Light One
return TRUE; // Initialization Went OK
}
```

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) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
```

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,0.0f,1.0f,0.0f) uses the variable yrot to rotate the cube on the y axis.

```
glTranslatef(0.0f,0.0f,z); // Translate Into/Out Of The Screen By z

glRotatef(xrot,1.0f,0.0f,0.0f); // Rotate On The X Axis By xrot
glRotatef(yrot,0.0f,1.0f,0.0f); // Rotate On The Y Axis By yrot
```

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]); // Select A Texture Based On filter
```

```
glBegin(GL_QUADS); // Start Drawing Quads
```

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

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

```
// Front Face
glNormal3f( 0.0f, 0.0f, 1.0f); // Normal Pointing Towards Viewer
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point 1 (Front)
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Point 2 (Front)
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Point 3 (Front)
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point 4 (Front)
// Back Face
glNormal3f( 0.0f, 0.0f,-1.0f); // Normal Pointing Away From Viewer
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point 1 (Back)
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point 2 (Back)
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Point 3 (Back)
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Point 4 (Back)
// Top Face
glNormal3f( 0.0f, 1.0f, 0.0f); // Normal Pointing Up
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point 1 (Top)
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point 2 (Top)
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Point 3 (Top)
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Point 4 (Top)
// Bottom Face
glNormal3f( 0.0f,-1.0f, 0.0f); // Normal Pointing Down
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point 1 (Bottom)
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Point 2 (Bottom)
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Point 3 (Bottom)
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point 4 (Bottom)
// Right face
glNormal3f( 1.0f, 0.0f, 0.0f); // Normal Pointing Right
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Point 1 (Right)
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Point 2 (Right)
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Point 3 (Right)
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Point 4 (Right)
// Left Face
glNormal3f(-1.0f, 0.0f, 0.0f); // Normal Pointing Left
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Point 1 (Left)
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Point 2 (Left)
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Point 3 (Left)
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Point 4 (Left)
glEnd(); // Done Drawing Quads
```

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; // Add xspeed To xrot
yrot+=yspeed; // Add yspeed To yrot
return TRUE; // Keep Going
}
```

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

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

```
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
if (keys['L'] && !lp) // L Key Being Pressed Not Held?
{
```

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 of false. So if we say light=!light, what we are actually saying is light equals NOT light. Which in english translates to if light equals true make light not true (false), and if light equals false, make light not false (true). So if light was true, it becomes false, and if light was false it becomes true.

```
lp=TRUE; // lp Becomes TRUE
light=!light; // Toggle Light TRUE/FALSE
```

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) // If Not Light
{
glDisable(GL_LIGHTING); // Disable Lighting
}
else // Otherwise
{
glEnable(GL_LIGHTING); // Enable 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']) // Has L Key Been Released?
{
lp=FALSE; // If So, lp Becomes FALSE
}
```

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) // Is F Key Being Pressed?
{
fp=TRUE; // fp Becomes TRUE
filter+=1; // filter Value Increases By One
if (filter>2) // Is Value Greater Than 2?
{
filter=0; // If So, Set filter To 0
}
}
if (!keys['F']) // Has F Key Been Released?
{
fp=FALSE; // If So, fp Becomes FALSE
}
```

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]) // Is Page Up Being Pressed?
{
z-=0.02f; // If So, Move Into The Screen
}
```

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 Page Down Being Pressed?
{
z+=0.02f; // If So, Move Towards The Viewer
}
```

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 Arrow Being Pressed?
{
xspeed-=0.01f; // If So, Decrease xspeed
}
if (keys[VK_DOWN]) // Is Down Arrow Being Pressed?
{
xspeed+=0.01f; // If So, Increase xspeed
}
if (keys[VK_RIGHT]) // Is Right Arrow Being Pressed?
{
yspeed+=0.01f; // If So, Increase yspeed
}
if (keys[VK_LEFT]) // Is Left Arrow Being Pressed?
{
yspeed-=0.01f; // If So, Decrease yspeed
}
```

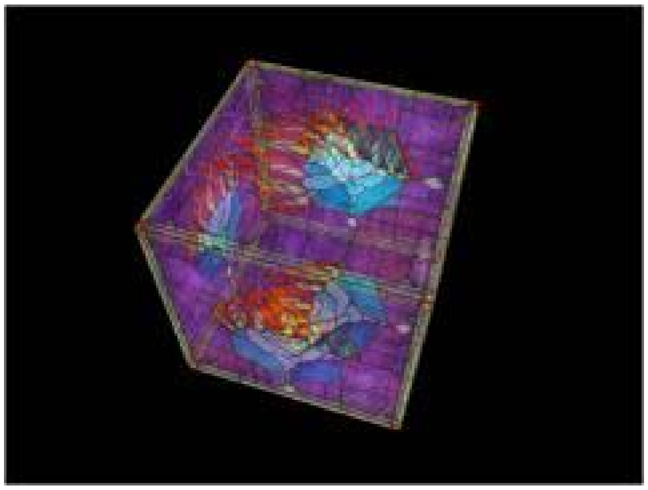
Like all the previous tutorials, make sure the title at the top of the window is correct.

```
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 / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Textures, Lighting & Keyboard Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
}
}
}

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

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**)

Lesson 08
Blending



Simple Transparency

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

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

The Blending Equation

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

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

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

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

This equation will yield transparent/translucent style effects.

Blending in OpenGL

We enable blending just like everything else. Then we set the equation, and turn off depth buffer writing when drawing transparent objects, since we still want objects behind the translucent shapes to be drawn. This isn't the proper way to blend, but most the time in simple projects it will work fine. **Rui Martins Adds:** The correct way is to draw all the transparent (with alpha < 1.0) polys after you have drawn the entire scene, and to draw them in reverse depth order (farthest first). This is due to the fact that blending two polygons (1 and 2) in different order gives different results, i.e. (assuming poly 1 is nearest to the viewer, the correct way would be to draw poly 2 first and then poly 1. If you look at it, like in reality, all the light comming from behind these two polys (which are transparent) has to pass poly 2 first and then poly 1 before it reaches the eye of the viewer. You should SORT THE TRANSPARENT POLYGONS BY DEPTH and draw them AFTER THE ENTIRE SCENE HAS BEEN DRAWN, with the DEPTH BUFFER ENABLED, or you will get incorrect results. I know this sometimes is a pain, but this is the correct way to do it.

We'll be using the code from the last tutorial. 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 Routine
bool active=TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
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: if (TextureImage[0]=LoadBMP("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.

```
if (TextureImage[0]=LoadBMP("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 ( NEW )
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Blending Function For Translucency Based On Source Alpha
Value ( NEW )
```

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 FALSE?
{
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().

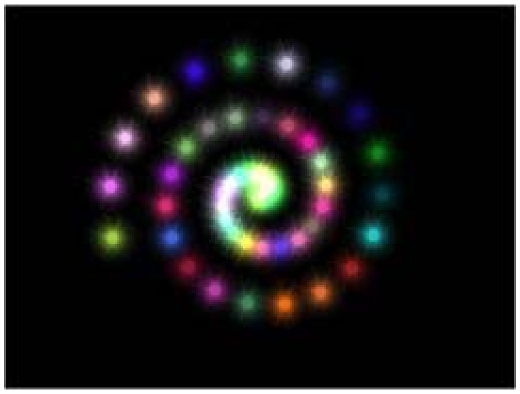
Questions?

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

Tom Stanis

Jeff Molofee (**NeHe**)

Lesson 09
Moving Bitmaps In 3D Space



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 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 Routine
bool active=TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
```

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

```
BOOL twinkle; // 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 To Draw
```

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 Structure For Star
{
int r, g, b; // Stars Color
GLfloat dist; // Stars Distance From Center
GLfloat angle; // Stars Current Angle
}
stars; // Structures Name Is Stars
stars star[num]; // Make 'star' Array Of 'num' Using Info From The Structure 'stars'
```

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 Away From Stars
GLfloat tilt=90.0f; // Tilt The View
GLfloat spin; // Spin Twinkling Stars

GLuint loop; // General Loop Variable
GLuint texture[1]; // Storage For One Texture

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 Filename Was Given
{
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 Pointer
}
return NULL; // If Load Failed Return NULL
}
```

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

```
int LoadGLTextures() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator

AUX_RGBImageRec *TextureImage[1]; // Create Storage Space For The Texture

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, 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 Memory
}

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

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 Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load Return FALSE
}

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 Calculations
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Set The Blending Function For Translucency
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 That Goes Through All The Stars
{
star[loop].angle=0.0f; // Start All The Stars At Angle Zero
```

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

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

```
star[loop].dist=(float(loop)/num)*5.0f; // Calculate Distance From The Center
star[loop].r=rand()%256; // Give star[loop] A Random Red Intensity
star[loop].g=rand()%256; // Give star[loop] A Random Green Intensity
star[loop].b=rand()%256; // Give star[loop] A Random Blue Intensity
}
return TRUE; // Initialization Went OK
}
```

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 Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Our Texture

for (loop=0; loop<num; loop++) // Loop Through All The Stars
{
glLoadIdentity(); // Reset The View Before We Draw Each Star
glTranslatef(0.0f,0.0f,zoom); // Zoom Into The Screen (Using The Value In 'zoom')
glRotatef(tilt,1.0f,0.0f,0.0f); // Tilt The View (Using The Value In 'tilt')
```

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 on the right, +z would be on the left, -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 second 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 Stars Angle
glTranslatef(star[loop].dist,0.0f,0.0f); // Move Forward On The X Plane
```

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. Then we need to cancel the screen tilt on the x axis. 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 Angle
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 Enabled
{
// 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 Textured Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd(); // Done Drawing The Textured Quad
}
```

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 On The Z Axis
// Assign A Color Using Bytes
glColor4ub(star[loop].r,star[loop].g,star[loop].b,255);
glBegin(GL_QUADS); // Begin Drawing The Textured Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd(); // Done Drawing The Textured Quad
```

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

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 Yet
{
star[loop].dist+=5.0f; // Move The Star 5 Units From The Center
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 OK
}
```

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 Buffering)
if (keys['T'] && !tp) // Is T Being Pressed And Is tp FALSE
{
tp=TRUE; // If So, Make tp TRUE
twinkle=!twinkle; // Make twinkle Equal The Opposite Of What It Is
}
```

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 Been Released
{
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 Up
}

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

if (keys[VK_PRIOR]) // Is Page Up Being Pressed
{
zoom-=0.2f; // Zoom Out
}

if (keys[VK_NEXT]) // Is Page Down Being Pressed
{
zoom+=0.2f; // Zoom In
}
```

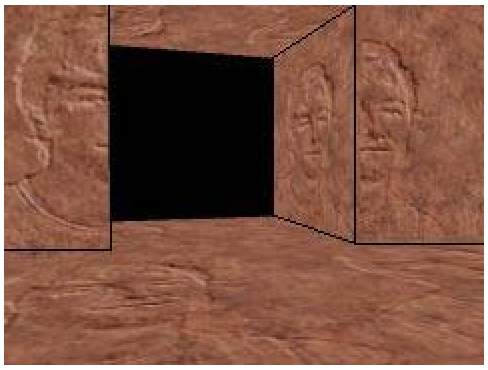
Like all the previous tutorials, make sure the title at the top of the window is correct.

```
if (keys[VK_F1]) // Is F1 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Textures, Lighting & Keyboard Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
}
}
}
```

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

Jeff Molofee (**NeHe**)

Lesson 10
Loading And Moving Through A 3D World



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

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

Data structure

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

```
typedef struct tagSECTOR // Build Our Sector Structure
{
int numtriangles; // Number Of Triangles In Sector
TRIANGLE* triangle; // Pointer To Array Of Triangles
} SECTOR; // Call It SECTOR
```

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

```
typedef struct tagTRIANGLE // Build Our Triangle Structure
{
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 catagory. The vertex holds the real data that OpenGL is interested in. We define each point on the triangle with it's position in 3D space (x, y, z) as well as it's texture coordinates (u, v).

```
typedef struct tagVERTEX // Build Our Vertex Structure
{
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 To Work With
filein = fopen(worldfile, "rt"); // Open Our File

...
(read our data)
...

fclose(filein); // Close Our File
return; // Jump Back
}
```

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 // Start A Loop
{
fgets(string, 255, f); // Read One Line
} while ((string[0] == '/') || (string[0] == '\n')); // See If It Is Worthy Of Processing
return; // Jump Back
}
```

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

NUMPOLLIES n

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

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

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

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 Right Arrow Being Pressed?
{
yrot -= 1.5f; // Rotate The Scene To The Left
}

if (keys[VK_LEFT]) // Is The Left Arrow Being Pressed?
{
yrot += 1.5f; // Rotate The Scene To The Right
}

if (keys[VK_UP]) // Is The Up Arrow Being Pressed?
{
xpos -= (float)sin(heading*piover180) * 0.05f; // Move On The X-Plane Based On Player Direction
zpos -= (float)cos(heading*piover180) * 0.05f; // Move On The Z-Plane Based On Player Direction
if (walkbiasangle >= 359.0f) // Is walkbiasangle>=359?
{
walkbiasangle = 0.0f; // Make walkbiasangle Equal 0
}
else // Otherwise
{
walkbiasangle+= 10; // If walkbiasangle < 359 Increase It By 10
}
walkbias = (float)sin(walkbiasangle * piover180)/20.0f; // Causes The Player To Bounce
}

if (keys[VK_DOWN]) // Is The Down Arrow Being Pressed?
{
xpos += (float)sin(heading*piover180) * 0.05f; // Move On The X-Plane Based On Player Direction
zpos += (float)cos(heading*piover180) * 0.05f; // Move On The Z-Plane Based On Player Direction
if (walkbiasangle <= 1.0f) // Is walkbiasangle<=1?
{
walkbiasangle = 359.0f; // Make walkbiasangle Equal 359
}
else // Otherwise
{
walkbiasangle-= 10; // If walkbiasangle > 1 Decrease It By 10
}
walkbias = (float)sin(walkbiasangle * piover180)/20.0f; // Causes The Player To Bounce
}
```

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

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

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

```
int DrawGLScene(GLvoid) // Draw The OpenGL Scene
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
glLoadIdentity(); // Reset The Current Matrix

GLfloat x_m, y_m, z_m, u_m, v_m; // Floating Point For Temp X, Y, Z, U And V Vertices
GLfloat xtrans = -xpos; // Used For Player Translation On The X Axis
GLfloat ztrans = -zpos; // Used For Player Translation On The Z Axis
GLfloat ytrans = -walkbias-0.25f; // Used For Bouncing Motion Up And Down
GLfloat sceneroty = 360.0f - yrot; // 360 Degree Angle For Player Direction

int numtriangles; // Integer To Hold The Number Of Triangles

glRotatef(lookupdown,1.0f,0,0); // Rotate Up And Down To Look Up And Down
glRotatef(sceneroty,0,1.0f,0); // Rotate Depending On Direction Player Is Facing
glTranslatef(xtrans, ytrans, ztrans); // Translate The Scene Based On Player Position
glBindTexture(GL_TEXTURE_2D, texture[filter]); // Select A Texture Based On filter

numtriangles = sector1.numtriangles; // Get The Number Of Triangles In Sector 1

// Process Each Triangle
for (int loop_m = 0; loop_m < numtriangles; loop_m++) // Loop Through All The Triangles
{
glBegin(GL_TRIANGLES); // Start Drawing Triangles
glNormal3f( 0.0f, 0.0f, 1.0f); // Normal Pointing Forward
x_m = sector1.triangle[loop_m].vertex[0].x; // X Vertex Of 1st Point
y_m = sector1.triangle[loop_m].vertex[0].y; // Y Vertex Of 1st Point
z_m = sector1.triangle[loop_m].vertex[0].z; // Z Vertex Of 1st Point
u_m = sector1.triangle[loop_m].vertex[0].u; // U Texture Coord Of 1st Point
v_m = sector1.triangle[loop_m].vertex[0].v; // V Texture Coord Of 1st Point
glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // Set The TexCoord And Vertice

x_m = sector1.triangle[loop_m].vertex[1].x; // X Vertex Of 2nd Point
y_m = sector1.triangle[loop_m].vertex[1].y; // Y Vertex Of 2nd Point
z_m = sector1.triangle[loop_m].vertex[1].z; // Z Vertex Of 2nd Point
u_m = sector1.triangle[loop_m].vertex[1].u; // U Texture Coord Of 2nd Point
v_m = sector1.triangle[loop_m].vertex[1].v; // V Texture Coord Of 2nd Point
glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // Set The TexCoord And Vertice

x_m = sector1.triangle[loop_m].vertex[2].x; // X Vertex Of 3rd Point
y_m = sector1.triangle[loop_m].vertex[2].y; // Y Vertex Of 3rd Point
z_m = sector1.triangle[loop_m].vertex[2].z; // Z Vertex Of 3rd Point
u_m = sector1.triangle[loop_m].vertex[2].u; // U Texture Coord Of 3rd Point
v_m = sector1.triangle[loop_m].vertex[2].v; // V Texture Coord Of 3rd Point
glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // Set The TexCoord And Vertice
glEnd(); // Done Drawing Triangles
}
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** (**ßetelgeuse**)*

***Jeff Molofee** (**NeHe**)*

Lesson 11
Flag Effect (Waving Texture)

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 The Points On The Grid Of Our "Wave"
int wiggle_count = 0; // Counter Used To Control How Fast Flag Waves
GLfloat hold; // Temporarily Holds A Floating Point Value
```

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 Lines
```

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(int x=0; x<45; x++)
{
// Loop Through The Y Plane
for(int y=0; y<45; y++)
{
// Apply The Wave To Our Mesh
points[x][y][0]=float((x/5.0f)-4.5f);
points[x][y][1]=float((y/5.0f)-4.5f);
points[x][y][2]=float(sin((((x/5.0f)*40.0f)/360.0f)*3.141592654*2.0f));
}
}
```

Thanks to Graham Gibbons for suggesting an integer loop to get rid of the spike in the ripple.

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. We use integer loops to prevent odd graphical glitches that appear when floating point calculations are used. We divide the x and y variables by 5 (i.e. 45 / 9 = 5) and subtract 4.5 from each of them to center the "wave". 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 Into Tiny Quads
```

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 Buffer
glLoadIdentity(); // Reset The Current Matrix

glTranslatef(0.0f,0.0f,-12.0f); // Translate 17 Units Into The Screen

glRotatef(xrot,1.0f,0.0f,0.0f); // Rotate On The X Axis
glRotatef(yrot,0.0f,1.0f,0.0f); // Rotate On The Y Axis
glRotatef(zrot,0.0f,0.0f,1.0f); // Rotate On The Z Axis

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 0-44 (45 Points)
{
for( y = 0; y < 44; y++ ) // Loop Through The Y Plane 0-44 (45 Points)
{
```

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 Value
float_y = float(y)/44.0f; // Create A Floating Point Y Value
float_xb = float(x+1)/44.0f; // Create A Floating Point Y Value+0.0227f
float_yb = float(y+1)/44.0f; // Create A Floating Point Y Value+0.0227f
```

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

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

glTexCoord2f( float_x, float_yb ); // Second Texture Coordinate (Top Left)
glVertex3f( points[x][y+1][0], points[x][y+1][1], points[x][y+1][2] );

glTexCoord2f( float_xb, float_yb ); // Third Texture Coordinate (Top Right)
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 (Bottom Right)
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 The Wave (Every 2nd Frame Only)
{
```

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 One Left Side Of Wave
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 The Far Left Stored Value
}
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 Variable
yrot+=0.2f; // Increase The Y Rotation Variable
zrot+=0.4f; // Increase The Z Rotation Variable

return TRUE; // Jump Back
}
```

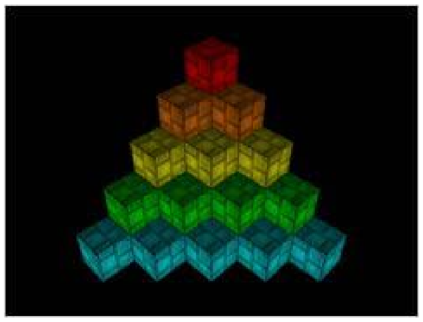
Standard NeHe rotation values. :) And that's it folks. Compile and you should have a nice rotating bitmapped "wave". I'm not sure what else to say except, whew.. This was LONG! But I hope you guys can follow it/get something out of it. If you have any questions, want me to clear something up or tell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

Jeff Molofee (**NeHe**)

Lesson 12
Display Lists



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> // 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 Routine
bool active=TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
```

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 List
GLuint top; // Storage For The Second Display List
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 BuildLists() // Build Box Display List
{
```

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

```
box=glGenLists(2); // Building Two Lists
```

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

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

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

```
glNewList(box,GL_COMPILE); // New Compiled box Display List
```

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

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

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

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

```
glBegin(GL_QUADS); // Start Drawing Quads
// Bottom Face
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
// Front Face
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Left Of The Texture and Quad
// Back Face
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Bottom Left Of The Texture and
Quad
// Right face
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
```

```
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Left Of The Texture and Quad
// Left Face
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // Bottom Left Of The Texture and
Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
glEnd(); // Done Drawing Quads
```

We tell OpenGL we're done making out list with the command glEndList(). Anything between glNewList() and glEndList is part of the Display List, anything before glNewList() or after glEndList() is not part of the current display list.

```
glEndList(); // Done Building The box List
```

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; // top List Value Is box List Value +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); // New Compiled top Display List
```

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); // Start Drawing Quad
// Top Face
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Bottom Right Of The Texture and Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
glEnd(); // Done Drawing Quad
```

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(); // Done Building The top Display List
}
```

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 Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load Return FALSE
}
BuildLists(); // Jump To The Code That Creates Our Display Lists
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
glEnable(GL_DEPTH_TEST); // Enables Depth Testing
glDepthFunc(GL_LEQUAL); // The Type Of Depth Testing To Do
```

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 Lighting (Assumes Light0 Is Set Up)
glEnable(GL_LIGHTING); // Enable Lighting
glEnable(GL_COLOR_MATERIAL); // Enable Material Coloring
```

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 Correction
return TRUE; // Initialization Went OK
```

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 Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
```

```
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 Y Plane
{
```

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 confussing, but it's actually not. On the X axis, the following happens:

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

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

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

```
// Position The Cubes On The Screen
glTranslatef(1.4f+(float(xloop)*2.8f)-(float(yloop)*1.4f),((6.0f-float(yloop))*2.4f)-7.0f,-
20.0f);
```

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 The Cubes Up And Down
glRotatef(45.0f+yrot,0.0f,1.0f,0.0f); // Spin Cubes Left And Right
```

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 Color
```

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 Color
```

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

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

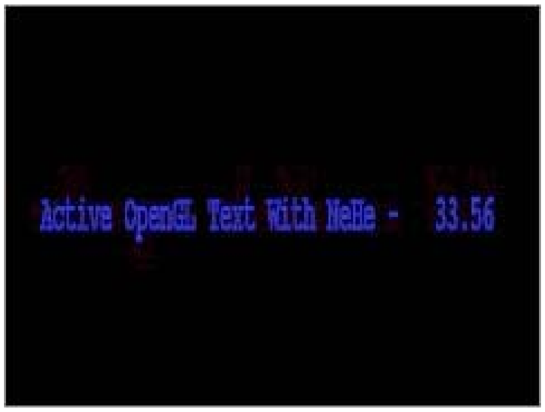
```
if (keys[VK_F1]) // Is F1 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Display List Tutorial",640,480,16,fullscreen))
{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

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**)

Lesson 13
Bitmap Fonts



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 avaialable. This tutorial will teach you how to use the exact same fonts in your own OpenGL programs. As a matter of fact... Any font you install on your computer can be used in your demos.

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

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

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

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

```
#include <windows.h> // Header File For Windows
#include <math.h> // Header File For Windows Math Library ( ADD )
#include <stdio.h> // Header File For Standard Input/Output ( ADD )
#include <stdarg.h> // Header File For Variable Argument Routines ( ADD )
#include <gl\gl.h> // Header File For The OpenGL32 Library
#include <gl\glu.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The 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. oldfont is used for good house keeping.

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
HFONT oldfont; // Used For Good House Keeping

base = glGenLists(96); // Storage For 96 Characters ( NEW )
```

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 at 0, 90, 180, and 270 degrees, the font usually gets cropped to fit inside it's invisible square border. Orientation Angle quoted from MSDN help Specifies the angle, in tenths of degrees, between each character's base line and the x-axis of the device. Unfortunately I have no idea what that means :(

```
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 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 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 trys to use a TRUETYPE Font.

```
OUT_TT_PRECIS, // Output Precision
```

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

```
CLIP_DEFAULT_PRECIS, // Clipping Precision
```

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 we just created. oldfont will point to the selected object. We then 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. After that we select the object oldfont points to and then we delete the font object.

```
oldfont = (HFONT)SelectObject(hDC, font); // Selects The Font We Want
wglUseFontBitmaps(hDC, 32, 96, base); // Builds 96 Characters Starting At Character 32
SelectObject(hDC, oldfont); // Selects The Font We Want
DeleteObject(font); // Delete The Font
}
```

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 List
{
glDeleteLists(base, 96); // Delete All 96 Characters ( NEW )
}
```

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 Of Arguments
```

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 Variables
vsprintf(text, fmt, ap); // And Converts Symbols To Actual Numbers
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 Bits ( NEW )
glListBase(base - 32); // Sets The Base Character to 32 ( NEW )
```

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

The line below does the following. First it tells OpenGL we're going to be displaying lists to the screen. strlen(text) finds out how many letters we're going to send to the screen. Next it needs to know what the largest list number were sending to it is going to be. We're not sending any more than 255 characters. The lists parameter is treated as an array of unsigned bytes, each in the range 0 through 255. Finally we tell it what to display by passing text (pointer to our string).

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

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

```
glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // Draws The Display List Text ( NEW )
glPopAttrib(); // Pops The Display List Bits ( NEW )
}
```

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 Calculations

BuildFont(); // Build The Font

return TRUE; // Initialization Went OK
}
```

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

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

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

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

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

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

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

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

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

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

Shawn T. sent me modified code that allows glPrint to pass variables to the screen. This means that you can increase a counter and display the results on the screen! It works like this... In the line below you see our normal text. Then there's a space, a dash, a space, then a "symbol" (%7.2f). Now you may look at %7.2f and say what the heck does that mean. It's very simple. % is like a marker saying don't print 7.2f to the screen, because it represents a variable. The 7 means a maximum of 7 digits will be displayed to the left of the decimal place. Then the decimal place, and right after the decimal place is a 2. The 2 means that only two digits will be displayed to the right of the decimal place. Finally, the f. The f means that the number we want to display is a floating point number. We want to display the value of cnt1 on the screen. As an example, if cnt1 was equal to 300.12345f the number we would end up seeing on the screen would be 300.12. The 3, 4, and 5 after the decimal place would be cut off because we only want 2 digits to appear after the decimal place.

I know if you're an experienced C programmer, this is absolute basic stuff, but there may be people out there that have never used printf. If you're interested in learning more about symbols, buy a book, or read through the MSDN.

```
glPrint("Active OpenGL Text With NeHe - %7.2f", cnt1); // Print GL Text To The Screen
```

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

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

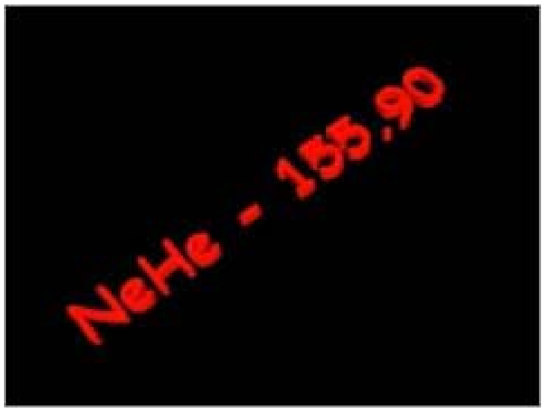
The last thing to do is add KillFont() to the end of KillGLWindow() just like I'm showing below. It's important to add this line. It cleans things up before we exit our program.

```
if (!UnregisterClass("OpenGL",hInstance)) // Are We Able To Unregister Class
{
MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
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**)

Lesson 14
Outline Fonts



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 Characters
```

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 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 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 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 trys to use a TRUETYPE Font.

```
OUT_TT_PRECIS, // Output Precision
```

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

```
CLIP_DEFAULT_PRECIS, // Clipping Precision
```

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 Created
```

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 DC
0, // Starting Character
255, // Number Of Display Lists To Build
base, // Starting Display Lists
```

That's not all however. We then set the deviation level. The closer to 0.0f, the smooth the font will look. After we set the deviation, we get to set the font thickness. This describes how thick the font is on the Z axis. 0.0f will produce a flat 2D looking font and 1.0f will produce a font with some depth.

The parameter WGL_FONT_POLYGONS tells OpenGL to create a solid font using polygons. If we use WGL_FONT_LINES instead, the font will be wireframe (made of lines). It's also important to note that if you use GL_FONT_LINES, normals will not be generated so lighting will not work properly.

The last parameter gmf points to the address buffer for the display list data.

```
0.0f, // Deviation From The True Outlines
0.2f, // Font Thickness In The Z Direction
WGL_FONT_POLYGONS, // Use Polygons, Not Lines
gmf); // Address Of Buffer To Recieve Data
}
```

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 Of The Text
char text[256]; // Holds Our String
va_list ap; // Pointer To List Of Arguments
```

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 Variables
vsprintf(text, fmt, ap); // And Converts Symbols To Actual Numbers
va_end(ap); // Results Are Stored In Text
```

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 5 letters length would equal 50 (5*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 Characters Width
}
```

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 Screen
```

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 Bits
glListBase(base); // Sets The Base Character to 0
```

Now that OpenGL knows where the characters are located, we can tell it to write the text to the screen. glCallLists writes the entire string of text to the screen at once by making multiple display list calls for you.

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

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

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

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

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 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 Calculations
glEnable(GL_LIGHT0); // Enable Default Light (Quick And Dirty) ( NEW )
glEnable(GL_LIGHTING); // Enable Lighting ( NEW )
glEnable(GL_COLOR_MATERIAL); // Enable Coloring Of Material ( NEW )

BuildFont(); // Build The Font ( ADD )

return TRUE; // Initialization Went OK
}
```

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 Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
```

```
glTranslatef(0.0f,0.0f,-10.0f); // Move Ten Units Into The Screen
```

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 Axis
glRotatef(rot*1.5f,0.0f,1.0f,0.0f); // Rotate On The Y Axis
glRotatef(rot*1.4f,0.0f,0.0f,1.0f); // Rotate On The Z Axis
```

Now for the crazy color cycling. As usual, I make use of the only variable that counts up (rot). The colors pulse up and down using COS and SIN. I divide the value of rot by different numbers so that each color isn't increasing at the same speed. The final results are nice.

```
// Pulsing Colors Based On The Rotation
glColor3f(1.0f*float(cos(rot/20.0f)),1.0f*float(sin(rot/25.0f)),1.0f-
0.5f*float(cos(rot/17.0f)));
```

My favorite part... Writing the text to the screen. I've used the same command we used to write Bitmap fonts to the screen. All you have to do to write the text to the screen is glPrint("{any text you want}"). It's that easy!

In the code below we'll print NeHe, a space, a dash, a space, and then whatever number is stored in rot divided by 50 (to slow down the counter a bit). If the number is larger that 999.99 the 4th digit to the left will be cut off (we're requesting only 3 digits to the left of the decimal place). Only 2 digits will be displayed after the decimal place.

```
glPrint("NeHe - %3.2f",rot/50); // Print GL Text To The Screen
```

Then we increase the rotation variable so the colors pulse and the text spins.

```
rot+=0.5f; // Increase The Rotation Variable
return TRUE; // Everything Went OK
}
```

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_ICONINFORMATION);
hInstance=NULL; // Set hInstance To NULL
}

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**)

Lesson 15
Texture Mapped Outline Fonts

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 Library
#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 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 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 Font Set

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 Characters
font = CreateFont( -12, // Height Of Font
0, // Width Of Font
0, // Angle Of Escapement
0, // Orientation Angle
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 Identifier ( Modified )
```

The next few lines have not changed.

```
OUT_TT_PRECIS, // Output Precision
CLIP_DEFAULT_PRECIS, // Clipping Precision
ANTIALIASED_QUALITY, // Output Quality
FF_DONTCARE|DEFAULT_PITCH, // Family And Pitch
```

Now that we've selected the symbol character set identifier, we can select the wingdings font!

```
"Wingdings"); // Font Name ( Modified )
```

The remaining lines of code have not changed.

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

wglUseFontOutlines( hDC, // Select The Current DC
0, // Starting Character
255, // Number Of Display Lists To Build
base, // Starting Display Lists
```

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 The True Outlines
```

The next three lines of code are still the same.

```
0.2f, // Font Thickness In The Z Direction
WGL_FONT_POLYGONS, // Use Polygons, Not Lines
gmf); // Address Of Buffer To Recieve Data
}
```

Right before ReSizeGLScene() we're going to add the following section of code to load our texture. You might recognize the code from previous tutorials. We create storage for the bitmap image. We load the bitmap image. We tell OpenGL to generate 1 texture, and we store this texture in texture[0].

I'm creating a mipmapped texture only because it looks better. The name of the texture is lights.bmp.

```
AUX_RGBImageRec *LoadBMP(char *Filename) // Loads A Bitmap Image
{
FILE *File=NULL; // File Handle

if (!Filename) // Make Sure A Filename Was Given
{
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 Pointer
}

return NULL; // If Load Failed Return NULL
}

int LoadGLTextures() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator

AUX_RGBImageRec *TextureImage[1]; // Create Storage Space For The Texture

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_UNSIGNED_BYTE, TextureImage[0]->data);
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 it's own. All you need to know is that GL_S and GL_T are texture coordinates. By default they are set up to take the current x location on the screen and the current y location on the screen and come up with a texture vertex. You'll notice the objects are not textured on the z plane... just stripes appear. The front and back faces are textured though, and that's all that matters. X (GL_S) will cover mapping the texture left to right, and Y (GL_T) will cover mapping the texture up and down.

GL_TEXTURE_GEN_MODE lets us select the mode of texture mapping we want to use on the S and T texture coordinates. You have 3 choices:

GL_EYE_LINEAR - The texture is fixed to the screen. It never moves. The object is mapped with whatever section of the texture it is passing over.

GL_OBJECT_LINEAR - This is the mode we are using. The texture is fixed to the object moving around the screen.

GL_SPHERE_MAP - Everyones favorite. Creates a metalic reflective type object.

It's important to note that I'm leaving out alot of code. We should be setting the GL_OBJECT_PLANE as well, but by default it's set to the parameters we want. Buy a good book if you're interested in learning more, or check out the MSDN help CD / DVD.

```
// Texturing Contour Anchored To The Object
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
// Texturing Contour Anchored To The Object
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glEnable(GL_TEXTURE_GEN_S); // Auto Texture Generation
glEnable(GL_TEXTURE_GEN_T); // Auto Texture Generation
}

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

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

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 Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load Return FALSE
}
BuildFont(); // Build The Font

glShadeModel(GL_SMOOTH); // Enable Smooth Shading
glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Black Background
glClearDepth(1.0f); // Depth Buffer Setup
glEnable(GL_DEPTH_TEST); // Enables Depth Testing
glDepthFunc(GL_LEQUAL); // The Type Of Depth Testing To Do
glEnable(GL_LIGHT0); // Quick And Dirty Lighting (Assumes Light0 Is Set Up)
glEnable(GL_LIGHTING); // Enable Lighting
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Calculations
```

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 Went OK
```

```
}
```

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

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth 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 Axis
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 Crossbones Symbol
rot+=0.1f; // Increase The Rotation Variable
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 Class
{
MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hInstance=NULL; // Set hInstance To NULL
}

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**)

Lesson 16
Cool Looking Fog

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 glClearColor() line has been modified to clear the screen to the same same color as the fog for a better effect. There isn't much code involved to make fog work. In all you will find this to be very simplistic.

```
glClearColor(0.5f,0.5f,0.5f,1.0f); // We'll Clear To The Color Of The Fog ( Modified )

glFogi(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.

Eric Desrosiers Adds: Little explanation of glHint(GL_FOG_HINT, hintval);

hintval can be : GL_DONT_CARE, GL_NICEST or GL_FASTEST

gl_dont_care - Lets opengl choose the kind of fog (per vertex of per pixel) and an unknown formula.
gl_nicest - Makes the fog per pixel (look good)
glfastest - Makes the fog per vertex (faster, but less nice)

The next line glFogf(GL_FOG_START, 1.0f); will establish how close to the screen the fog should start. You can change the number to whatever you want depending on where you want the fog to start. The next line is similar, glFogf(GL_FOG_END, 5.0f);. This tells the OpenGL program how far into the screen the fog should go.

Keypress Events

Now that we've setup the fog drawing code we will add the keyboard commands to cycle through the different fog modes. This code goes down at the end of the program with all the other key handling code.

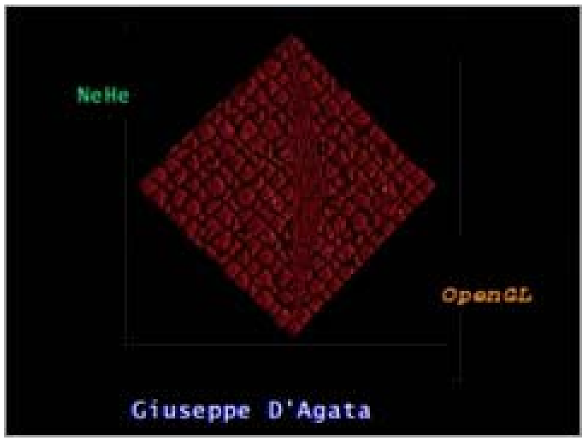
```
if (keys['G'] && !gp) // Is The G Key Being Pressed?
{
gp=TRUE; // gp Is Set To TRUE
fogfilter+=1; // Increase fogfilter By One
if (fogfilter>2) // Is fogfilter Greater Than 2?
{
fogfilter=0; // If So, Set fogfilter To Zero
}
glFogi (GL_FOG_MODE, fogMode[fogfilter]); // Fog Mode
}
if (!keys['G']) // 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**)

Jeff Molofee (**NeHe**)

Lesson 17
2D Texture Font



This tutorial brought to you by NeHe & Giuseppe D'Agata...

I know everyones probably sick of fonts. The text tutorials I've done so far not only display text, they display 3D text, texture mapped text, and can handle variables. But what happens if you're porting your project to a machine that doesn't support Bitmap or Outline fonts?

Thanks to Giuseppe D'Agata we have yet another font tutorial. What could possibly be left you ask!? If you remember in the first Font tutorial I mentioned using textures to draw letters to the screen. Usually when you use textures to draw text to the screen you load up your favorite art program, select a font, then type the letters or phase you want to display. You then save the bitmap and load it into your program as a texture. Not very efficient for a program that require alot of text, or text that continually changes!

This program uses just ONE texture to display any of 256 different characters on the screen. Keep in mind your average character is just 16 pixels wide and roughly 16 pixels tall. If you take your standard 256x256 texture it's easy to see that you can fit 16 letters across, and you can have a total of 16 rows up and down. If you need a more detailed explanation: The texture is 256 pixels wide, a character is 16 pixels wide. 256 divided by 16 is 16 :)

So... Lets create a 2D textured font demo! This program expands on the code from lesson 1. In the first section of the program, we include the math and stdio libraries. We need the math library to move our letters around the screen using SIN and COS, and we need the stdio library to make sure the bitmaps we want to use actually exist before we try to make textures out of them.

```
#include <windows.h> // Header File For Windows
#include <math.h> // Header File For Windows Math Library ( ADD )
#include <stdio.h> // Header File For Standard Input/Output ( ADD )
#include <gl\gl.h> // Header File For The OpenGL32 Library
#include <gl\glu.h> // Header File For The GLu32 Library
#include <gl\glaux.h> // Header File For The 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); // Declaration For WndProc
```

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 A Bitmap Image
{
FILE *File=NULL; // File Handle
if (!Filename) // Make Sure A Filename Was Given
{
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 Pointer
}
return NULL; // If Load Failed Return NULL
}
```

The follwing code has also changed very little from the code used in previous tutorials. If you're not sure what each of the following lines do, go back and review.

Note that TextureImage[ ] is going to hold 2 rgb image records. It's very important to double check code that deals with loading or storing our textures. One wrong number could result in a memory leak or crash!

```
int LoadGLTextures() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator
AUX_RGBImageRec *TextureImage[2]; // Create Storage Space For The Textures
```

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 The Pointer To NULL

if ((TextureImage[0]=LoadBMP("Data/Font.bmp")) && // Load The Font Bitmap
(TextureImage[1]=LoadBMP("Data/Bumps.bmp"))) // Load The Texture Bitmap
{
Status=TRUE; // Set The Status To TRUE
```

Another important line to double check. I can't begin to tell you how many emails I've received from people asking *"why am I only seeing one texture, or why are my textures all white!?!"*. Usually this line is the problem. If you were to replace the 2 with a 1, only one texture would be created and the second texture would appear all white. If you replaced the 2 with a 3 you're program may crash!

You should only have to call glGenTextures() once. After glGenTextures() you should generate all your textures. I've seen people put a glGenTextures() line before each texture they create. Usually they causes the new texture to overwrite any textures you've already created. It's a good idea to decide how many textures you need to build, call glGenTextures() once, and then build all the textures. It's not wise to put glGenTextures() inside a loop unless you have a reason to.

```
glGenTextures(2, &texture[0]); // Create Two Texture

for (loop=0; loop<2; loop++) // Loop Through All The Textures
{
// 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, TextureImage[loop]->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
}
}
```

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

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 Our Font Display List
{
```

The following two variable will be used to hold the position of each letter inside the font texture. cx will hold the position from left to right inside the texture, and cy will hold the position up and down.

```
float cx; // Holds Our X Character Coord
float cy; // Holds Our Y Character Coord
```

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); // Creating 256 Display Lists
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Our Font Texture
```

Now we start our loop. The loop will build all 256 characters, storing each character in it's own display lists.

```
for (loop=0; loop<256; loop++) // Loop Through All 256 Lists
{
```

The first line below may look a little puzzling. The % symbol means the remainder after loop is divided by 16. cx will move us through the font texture from left to right. You'll notice later in the code we subtract cy from 1 to move us from top to bottom instead of bottom to top. The % symbol is fairly hard to explain but I will make an attempt.

All we are really concerned about is (loop%16) the /16.0f just converts the results into texture coordinates. So if loop was equal to 16... cx would equal the remained of 16/16 which would be 0. but cy would equal 16/16 which is 1. So we'd move down the height of one character, and we wouldn't move to the right at all. Now if loop was equal to 17, cx would be equal to 17/16 which would be 1.0625. The remainder .0625 is also equal to 1/16th. Meaning we'd move 1 character to the right. cy would still be equal to 1 because we are only concerned with the number to the left of the decimal. 18/16 would gives us 2 over 16 moving us 2 characters to the right, and still one character down. If loop was 32, cx would once again equal 0, because there is no remained when you divide 32 by 16, but cy would equal 2. Because the number to the left of the decimal would now be 2, moving us down 2 characters from the top of our font texture. Does that make sense?

```
cx=float(loop%16)/16.0f; // X Position Of Current Character
cy=float(loop/16)/16.0f; // Y Position Of Current Character
```

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 Building A List
```

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 Quad For Each Character
```

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); // Texture Coord (Bottom Left)
glVertex2i(0,0); // Vertex Coord (Bottom Left)
```

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); // Texture Coord (Bottom Right)
glVertex2i(16,0); // Vertex Coord (Bottom Right)
```

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); // Texture Coord (Top Right)
glVertex2i(16,16); // Vertex Coord (Top Right)
```

Finally we move left to set our last texture coordinate at the top left of our character.

```
glTexCoord2f(cx,1-cy); // Texture Coord (Top Left)
glVertex2i(0,16); // Vertex Coord (Top Left)
glEnd(); // Done Building Our Quad (Character)
```

Finally, we translate 10 pixels to the right, placing us to the right of our texture. If we didn't translate, the letters would all be drawn on top of eachother. Because our font is so narrow, we don't want to move 16 pixels to the right. If we did, there would be big spaces between each letter. Moving by just 10 pixels eliminates the spaces.

```
glTranslated(10,0,0); // Move To The Right Of The Character
glEndList(); // Done Building The Display List
} // Loop Until All 256 Are Built
}
```

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) // Delete The Font From Memory
{
glDeleteLists(base,256); // Delete All 256 Display Lists
}
```

The next section of code is where all of our drawing is done. Everything is fairly new so I'll try to explain each line in great detail. Just a small note: Alot can be added to this code, such as variable support, character sizing, spacing, and alot of checking to restore things to how they were before we decided to print.

glPrint() takes three parameters. The first is the x position on the screen (the position from left to right). Next is the y position on the screen (up and down... 0 at the bottom, bigger numbers at the top). Then we have our actual string (the text we want to print), and finally a variable called set. If you have a look at the bitmap that Giuseppe D'Agata has made, you'll notice there are two different character sets. The first character set is normal, and the second character set is italicized. If set is 0, the first character set is selected. If set is 1 or greater the second character set is selected.

```
GLvoid glPrint(GLint x, GLint y, char *string, int set) // Where The Printing Happens
{
```

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) // Is set Greater Than One?
{
set=1; // If So, Make Set Equal One
}
```

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 Our Font Texture
```

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); // Disables Depth Testing
```

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); // Select The Projection Matrix
glPushMatrix(); // Store The Projection Matrix
```

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. There is no depth so we set the z values to -1 & 1.

```
glLoadIdentity(); // Reset The Projection Matrix
glOrtho(0,640,0,480,-1,1); // Set Up An Ortho Screen
```

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 The Modelview Matrix
glPushMatrix(); // Store The Modelview Matrix
glLoadIdentity(); // Reset The Modelview Matrix
```

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); // Position The Text (0,0 - Bottom Left)
```

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)); // Choose The Font Set (0 or 1)
```

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 The Text To The Screen
```

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); // Select The Projection Matrix
glPopMatrix(); // Restore The Old Projection Matrix
```

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 The Modelview Matrix
glPopMatrix(); // Restore The Old Projection Matrix
```

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); // Enables Depth Testing
}
```

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

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
```

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()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load 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(); // Build The Font
```

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); // Clear The Background Color To Black
glClearDepth(1.0); // Enables Clearing Of The Depth Buffer
glDepthFunc(GL_LEQUAL); // The Type Of Depth Test To Do
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Select The Type Of Blending
glShadeModel(GL_SMOOTH); // Enables Smooth Color Shading
glEnable(GL_TEXTURE_2D); // Enable 2D Texture Mapping
return TRUE; // Initialization Went OK
}
```

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) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The Modelview Matrix
```

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 Our Second Texture
glTranslatef(0.0f,0.0f,-5.0f); // Move Into The Screen 5 Units
glRotatef(45.0f,0.0f,0.0f,1.0f); // Rotate On The Z Axis 45 Degrees (Clockwise)
```

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); // Rotate On The X & Y Axis By cnt1 (Left To Right)
```

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); // Disable Blending Before We Draw In 3D
glColor3f(1.0f,1.0f,1.0f); // Bright White
glBegin(GL_QUADS); // Draw Our First Texture Mapped Quad
glTexCoord2d(0.0f,0.0f); // First Texture Coord
glVertex2f(-1.0f, 1.0f); // First Vertex
glTexCoord2d(1.0f,0.0f); // Second Texture Coord
glVertex2f( 1.0f, 1.0f); // Second Vertex
glTexCoord2d(1.0f,1.0f); // Third Texture Coord
glVertex2f( 1.0f,-1.0f); // Third Vertex
glTexCoord2d(0.0f,1.0f); // Fourth Texture Coord
glVertex2f(-1.0f,-1.0f); // Fourth Vertex
glEnd(); // Done Drawing The First Quad
```

Immediately after we've drawn the first quad, we rotate 90 degrees on both the x axis and y axis. We then draw another quad. The second quad cuts through the middle of the first quad, creating a nice looking shape.

```
glRotatef(90.0f,1.0f,1.0f,0.0f); // Rotate On The X & Y Axis By 90 Degrees (Left To Right)
glBegin(GL_QUADS); // Draw Our Second Texture Mapped Quad
glTexCoord2d(0.0f,0.0f); // First Texture Coord
glVertex2f(-1.0f, 1.0f); // First Vertex
glTexCoord2d(1.0f,0.0f); // Second Texture Coord
glVertex2f( 1.0f, 1.0f); // Second Vertex
glTexCoord2d(1.0f,1.0f); // Third Texture Coord
glVertex2f( 1.0f,-1.0f); // Third Vertex
glTexCoord2d(0.0f,1.0f); // Fourth Texture Coord
glVertex2f(-1.0f,-1.0f); // Fourth Vertex
glEnd(); // Done Drawing Our Second Quad
```

After both texture mapped quads have been drawn, we enable enable blending, and draw our text.

```
glEnable(GL_BLEND); // Enable Blending
glLoadIdentity(); // Reset The View
```

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+cnt2)));
```

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 GL Text To The
Screen

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

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

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 Offset Text To The
Screen
```

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; // Increase The First Counter
cnt2+=0.0081f; // Increase The Second Counter
return TRUE; // Everything Went OK
}
```

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 Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed 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,fullscreen))
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit Received?
{
done=TRUE; // ESC or DrawGLScene Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
}
}
}

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

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.

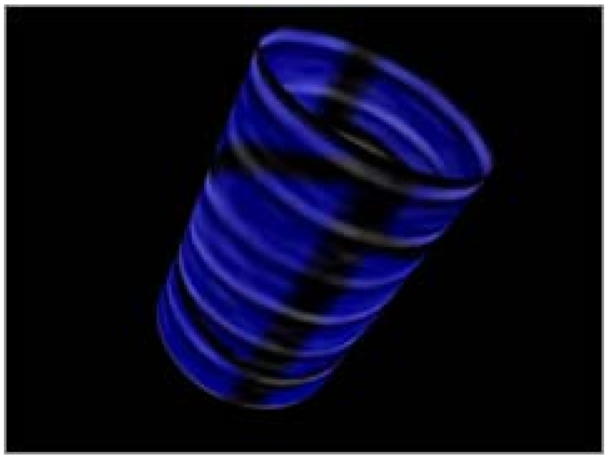
I'd like to thank Giuseppe D'Agata for the original version of this tutorial. I've modified it heavily, and converted it to the new base code, but without him sending me the code I probably wouldn't have written the tutorial. His version of the code had a few more options, such as spacing the characters, 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

Jeff Molofee (**NeHe**)

Lesson 18
Quadrics



Quadrics

Quadrics are a way of drawing complex objects that would usually take a few FOR loops and some background in trigonometry.

We'll be using the code from lesson seven. We will add 7 variables and modify the texture to add some variety :)

```
#include <windows.h> // Header File For Windows
#include <stdio.h> // Header File For Standard Input/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 Routine
bool active=TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
bool light; // Lighting ON/OFF
bool lp; // L Pressed?
bool fp; // F Pressed?
bool sp; // Spacebar Pressed? ( NEW )

int part1; // Start Of Disc ( NEW )
int part2; // End Of Disc ( NEW )
int p1=0; // Increase 1 ( NEW )
int p2=1; // Increase 2 ( 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

GLUquadricObj *quadratic; // Storage For Our Quadratic Objects ( NEW )

GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f }; // Ambient Light Values
GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f }; // Diffuse Light Values
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f }; // Light Position

GLuint filter; // Which Filter To Use
GLuint texture[3]; // Storage for 3 textures
GLuint object=0; // Which Object To Draw ( NEW )

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

Okay now move down to InitGL(), We're going to add 3 lines of code here to initialize our quadratic. Add these 3 lines after you enable light1 but before you return true. The first line of code initializes the Quadratic and creates a pointer to where it will be held in memory. If it can't be created it returns 0. The second line of code creates smooth normals on the quadratic so lighting will look great. Other possible values are GLU_NONE, and GLU_FLAT. Last we enable texture mapping on our quadratic. Texture mapping is kind of awkward and never goes the way you planned as you can tell from the crate texture.

```
quadratic=gluNewQuadric(); // Create A Pointer To The Quadric Object ( NEW )
gluQuadricNormals(quadratic, GLU_SMOOTH); // Create Smooth Normals ( NEW )
gluQuadricTexture(quadratic, GL_TRUE); // Create Texture Coords ( NEW )
```

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 Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Left Of The Texture and Quad
// 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 Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Bottom Left Of The Texture and
Quad
// 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 Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Bottom Right Of The Texture and Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
// 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 Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
// 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 Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // Bottom Left Of The Texture and Quad
// 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 Left Of The Texture and
Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // Bottom Right Of The Texture and
Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // Top Left Of The Texture and Quad
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 Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
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 Axis
glRotatef(yrot,0.0f,1.0f,0.0f); // Rotate On The Y Axis

glBindTexture(GL_TEXTURE_2D, texture[filter]); // Select A Filtered Texture

// This Section Of Code Is New ( NEW )
switch(object) // Check object To Find Out What To Draw
```

```
{
case 0: // Drawing Object 1
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 fouth parameter (32) is how many subdivisions there are "around" the Z axis, and finally, the fifth parameter (32) is the amount of subdivisions "along" the Z axis. The more subdivisions there are the more detailed the object is. By increase the amount of subdivisions you add more polygons to the object. So you end up sacrificing speed for quality. Most of the time it's easy to find a happy medium.

```
case 1: // Drawing Object 2
glTranslatef(0.0f,0.0f,-1.5f); // Center The Cylinder
gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // Draw Our Cylinder
break; // Done
```

The third object we create will be a CD shaped disc. The first parameter (0.5f) is the inner radius of the disk. This value can be zero, meaning there will be no hole in the middle. The larger the inner radius is, the bigger the hole in the middle of the disc will be. The second parameter (1.5f) is the outer radius. This value should be larger than the inner radius. If you make this value a little bit larger than the inner radius you will end up with a thing ring. If you make this value alot larger than the inner radius you will end up with a thick ring. The third parameter (32) is the number of slices that make up the disc. Think of slices like the slices in a pizza. The more slices you have, the smoother the outer edge of the disc will be. Finally the fourth parameter (32) is the number of rings that make up the disc. The rings are are similar to the tracks on a record. Circles inside circles. These ring subdivide the disc from the inner radius to the outer radius, adding more detail. Again, the more subdivisions there are, the slow it will run.

```
case 2: // Drawing Object 3
gluDisk(quadratic,0.5f,1.5f,32,32); // Draw A Disc (CD Shape)
break; // Done
```

Our fourth object is an object that I know many of you have been dying to figure out. The Sphere! This one is quite simple. The first parameter is the radius of the sphere. In case you're not familiar with radius/diameter, etc, the radius is the distance from the center of the object to the outside of the object. In this case our radius is 1.3f. Next we have our subdivision "around" the Z axis (32), and our subdivision "along" the Z axis (32). The more subdivisions you have the smoother the sphere will look. Spheres usually require quite a few subdivisions to make them look smooth.

```
case 3: // Drawing Object 4
gluSphere(quadratic,1.3f,32,32); // Draw A Sphere
break; // Done
```

Our fifth object is created using the same command that we used to create a Cylinder. If you remember, when we were creating the Cylinder the first two parameters controlled the radius of the cylinder at the bottom and the top. To make a cone it makes sense that all we'd have to do is make the radius at one end Zero. This will create a point at one end. So in the code below, we make the radius at the top of the cylinder equal zero. This creates our point, which also creates our cone.

```
case 4: // Drawing Object 5
glTranslatef(0.0f,0.0f,-1.5f); // Center The Cone
gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32); // A Cone With A Bottom Radius Of .5 And A Height
Of 2
break; // Done
```

Our sixth object is created with gluPartialDisc. The object we create using this command will look exactly like the disc we created above, but with the command gluPartialDisk there are two new parameters. The fifth parameter (part1) is the start angle we want to start drawing the disc at. The sixth parameter is the sweep angle. The sweep angle is the distance we travel from the current angle. We'll increase the sweep angle, which causes the disc to be slowly drawn to the screen in a clockwise direction. Once our sweep hits 360 degrees we start to increase the start angle. the makes it appear as if the disc is being erased, then we start all over again!

```
case 5: // Drawing Object 6
part1+=p1; // Increase Start Angle
part2+=p2; // Increase Sweep Angle

if(part1>359) // 360 Degrees
{
p1=0; // Stop Increasing Start Angle
part1=0; // Set Start Angle To Zero
p2=1; // Start Increasing Sweep Angle
part2=0; // Start Sweep Angle At Zero
}
if(part2>359) // 360 Degrees
{
p1=1; // Start Increasing Start Angle
p2=0; // Stop Increasing Sweep Angle
}
gluPartialDisk(quadratic,0.5f,1.5f,32,32,part1,part2-part1); // A Disk Like The One Before
break; // Done
};

xrot+=xspeed; // Increase Rotation On X Axis
yrot+=yspeed; // Increase Rotation On Y Axis
return TRUE; // Keep Going
}
```

In the KillGLWindow() section of code, we need to delete the quadratic to free up system resources. We do this with the command gluDeleteQuadratic.

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
{
gluDeleteQuadric(quadratic); // Delete Quadratic - Free Resources
```

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

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

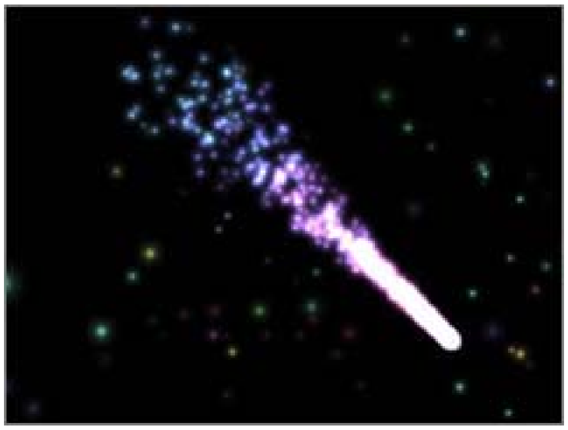
Thats all! Now you can draw quadrics in OpenGL. Some really impressive things can be done with morphing and quadrics. The animated disc is an example of simple morphing.

Everyone if you have time go check out my website, TipTup.Com 2000.

GB Schmick (**TipTup**)

Jeff Molofee (**NeHe**)

Lesson 19
Particle Engine Using Triangle Strips



Welcome to Tutorial 19. You've learned alot, and now you want to play. I will introduce one new command in this tutorial... The triangle strip. It's very easy to use, and can help speed up your programs when drawing alot of triangles.

In this tutorial I will teach you how to make a semi-complex Particle Engine. Once you understand how particle engines work, creating effects such as fire, smoke, water fountains and more will be a piece of cake!

I have to warn you however! Until today I had never written a particle engine. I had this idea that the 'famous' particle engine was a very complex piece of code. I've made attempts in the past, but usually gave up after I realized I couldn't control all the points without going crazy.

You might not believe me when I tell you this, but this tutorial was written 100% from scratch. I borrowed no ones ideas, and I had no technical information sitting in front of me. I started thinking about particles, and all of a sudden my head filled with ideas (brain turning on?). Instead of thinking about each particle as a pixel that had to go from point 'A' to point 'B', and do this or that, I decided it would be better to think of each particle as an individual object responding to the environment around it. I gave each particle life, random aging, color, speed, gravitational influence and more.

Soon I had a finished project. I looked up at the clock and realized aliens had come to get me once again. Another 4 hours gone! I remember stopping now and then to drink coffee and blink, but 4 hours... ?

So, although this program in my opinion looks great, and works exactly like I wanted it to, it may not be the proper way to make a particle engine. I don't care personally, as long as it works well, and I can use it in my projects! If you are the type of person that needs to know you're conforming, then spend hours browsing the net looking for information. Just be warned. The few code snippits you do find may appear cryptic :)

This tutorial uses the base code from lesson 1. There is alot of new code however, so I'll rewrite any section of code that contains changes (makes it easier to understand).

Using the code from lesson 1, we'll add 5 new lines of code at the top of our program. The first line (stdio.h) allows us to read data from files. It's the same line we've added to previous tutorials the use texture mapping. The second line defines how many particles were going to create and display on the screen. Define just tells our program that MAX_PARTICLES will equal whatever value we specify. In this case 1000. The third line will be used to toggle 'rainbow mode' off and on. We'll set it to on by default. sp and rp are variables we'll use to prevent the spacebar or return key from rapidly repeating when held down.

```
#include <windows.h> // 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 Of Tail)
float yspeed; // Base Y Speed (To Allow Keyboard Direction Of Tail)
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 Particle
{
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 postive it will move towards the viewer.

```
float xi; // X Direction
float yi; // Y Direction
float zi; // Z Direction
```

Lastly, 3 more variables! Each of these variables can be thought of as gravity. If xg is a positive value, our particle will pull to the right. If it's negative our particle will be pulled to the left. So if our particle is moving left (negative) and we apply a positive gravity, the speed will eventually slow so much that our particle will start moving the opposite direction. yg pulls up or down and zg pulls towards or away from the viewer.

```
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 Info)
```

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 0 to 11 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 Given
{
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 Pointer
}
return NULL; // If Load Failed Return NULL
}
```

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

```
int LoadGLTextures() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator

AUX_RGBImageRec *TextureImage[1]; // Create Storage Space For The Texture

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 Textures

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 Memory
}
free(TextureImage[0]); // Free The Image Structure
}
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 GL Window
{
if (height==0) // Prevent A Divide By Zero By
{
height=1; // Making Height Equal One
}

glViewport(0, 0, width, height); // Reset The Current Viewport

glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
glLoadIdentity(); // Reset The Projection Matrix

// 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 Setup For OpenGL Goes Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load Return FALSE
}
```

We enable smooth shading, clear our background to black, disable depth testing, blending and texture mapping. After enabling texture mapping we select our particle texture.

```
glShadeModel(GL_SMOOTH); // Enables Smooth Shading
glClearColor(0.0f,0.0f,0.0f,0.0f); // Black Background
glClearDepth(1.0f); // Depth Buffer Setup
glDisable(GL_DEPTH_TEST); // Disables Depth Testing
glEnable(GL_BLEND); // Enable Blending
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Type Of Blending To Perform
glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST); // Really Nice Perspective Calculations
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST); // Really Nice Point Smoothing
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glBindTexture(GL_TEXTURE_2D,texture[0]); // Select Our 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++) // Initials All The Textures
{
particle[loop].active=true; // Make All The Particles Active
particle[loop].life=1.0f; // Give All The Particles 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 multiply it by the number of colors in our color table divided by the maximum number of particles (MAX_PARTICLES). This prevents the final color value from being higher than our max number of colors (12).

Some quick examples: 900*(12/900)=12. 1000*(12/1000)=12, etc.

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

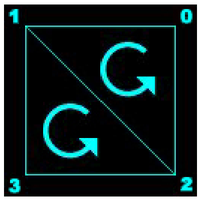
Now we'll set the direction that each particle moves, along with the speed. We're going to multiply the results by 10.0f to create a spectacular explosion when the program first starts.

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 Speed On X Axis
particle[loop].yi=float((rand()%50)-25.0f)*10.0f; // Random Speed On Y Axis
```

```
particle[loop].zi=float((rand()%50)-25.0f)*10.0f; // Random Speed On Z Axis
```

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; // Set Horizontal Pull To Zero
particle[loop].yg=-0.8f; // Set Vertical Pull Downward
particle[loop].zg=0.0f; // Set Pull On Z Axis To Zero
}
return TRUE; // Initialization Went OK
}
```

Now for the fun stuff. The next section of code is where we draw the particle, check for gravity, etc. It's important that you understand what's going on, so please read carefully :)

We reset the Modelview Matrix only once. We'll position the particles using the glVertex3f() command instead of using tranlations, that way we don't alter the modelview matrix while drawing our particles.

```
int DrawGLScene(GLvoid) // Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
glLoadIdentity(); // Reset The ModelView Matrix
```

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

```
for (loop=0;loop<MAX_PARTICLES;loop++) // Loop Through All The Particles
{
```

First thing we do is check to see if the particle is active. If it's not active, it wont be updated. In this program they're all active, all the time. But in a program of your own, you may want to make certain particles inactive.

```
if (particle[loop].active) // If The Particle Is Active
{
```

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 Our Particle X Position
float y=particle[loop].y; // Grab Our Particle Y Position
float z=particle[loop].z+zoom; // Particle Z Pos + 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 1.0f. If you need the particles to burn longer, try reducing the fade speed so that the particle doesn't fade out as fast.

```
// Draw The Particle Using Our RGB Values, Fade The Particle Based On It's Life
glColor4f(particle[loop].r,particle[loop].g,particle[loop].b,particle[loop].life);
```

We have the particle position and the color is set. All that we have to do now is draw our particle. Instead of using a textured quad, I've decided to use a textured triangle strip to speed the program up a bit. Most 3D cards can draw triangles alot faster than they can draw quads. Some 3D cards will convert the quad to two triangles for you, but some don't. So we'll do the work ourselves. We start off by telling OpenGL we want to draw a triangle strip.

```
glBegin(GL_TRIANGLE_STRIP); // Build Quad From A Triangle Strip
```

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 <u>not</u> 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 Right
glTexCoord2d(0,1); glVertex3f(x-0.5f,y+0.5f,z); // Top Left
glTexCoord2d(1,0); glVertex3f(x+0.5f,y-0.5f,z); // Bottom Right
glTexCoord2d(0,0); glVertex3f(x-0.5f,y-0.5f,z); // Bottom Left
```

Finally we tell OpenGL that we are done drawing our triangle strip.

```
glEnd(); // Done Building Triangle Strip
```

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 On The X Axis By X Speed
particle[loop].y+=particle[loop].yi/(slowdown*1000); // Move On The Y Axis By Y Speed
particle[loop].z+=particle[loop].zi/(slowdown*1000); // Move On The Z Axis By Z Speed
```

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; // Take Pull On X Axis Into Account
particle[loop].yi+=particle[loop].yg; // Take Pull On Y Axis Into Account
particle[loop].zi+=particle[loop].zg; // Take Pull On Z Axis Into Account
```

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 Particles Life By 'Fade'
```

Now we check to see if the particle is still alive after having life taken from it.

```
if (particle[loop].life<0.0f) // If Particle Is Burned Out
{
```

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; // Give It New Life
particle[loop].fade=float(rand()%100)/1000.0f+0.003f; // Random Fade Value
```

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; // Center On X Axis
particle[loop].y=0.0f; // Center On Y Axis
particle[loop].z=0.0f; // Center On Z Axis
```

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); // X Axis Speed And Direction
particle[loop].yi=yspeed+float((rand()%60)-30.0f); // Y Axis Speed And Direction
particle[loop].zi=float((rand()%60)-30.0f); // Z Axis Speed And Direction
```

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]; // Select Red From Color Table
particle[loop].g=colors[col][1]; // Select Green From Color Table
particle[loop].b=colors[col][2]; // Select Blue From Color Table
}
```

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 Upwards
if (keys[VK_NUMPAD8] && (particle[loop].yg<1.5f)) particle[loop].yg+=0.01f;
```

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 Pull Downwards
if (keys[VK_NUMPAD2] && (particle[loop].yg>-1.5f)) particle[loop].yg-=0.01f;
```

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 Right
if (keys[VK_NUMPAD6] && (particle[loop].xg<1.5f)) particle[loop].xg+=0.01f;
```

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 Pull Left
if (keys[VK_NUMPAD4] && (particle[loop].xg>-1.5f)) particle[loop].xg-=0.01f;
```

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]) // Tab Key Causes A Burst
{
particle[loop].x=0.0f; // Center On X Axis
particle[loop].y=0.0f; // Center On Y Axis
particle[loop].z=0.0f; // Center On Z Axis
particle[loop].xi=float((rand()%50)-26.0f)*10.0f; // Random Speed On X Axis
particle[loop].yi=float((rand()%50)-25.0f)*10.0f; // Random Speed On Y Axis
particle[loop].zi=float((rand()%50)-25.0f)*10.0f; // Random Speed On Z Axis
}
}
}
return TRUE; // Everything Went OK
}
```

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 hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}

// Create Our OpenGL Window
if (!CreateGLWindow("NeHe's Particle Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
```

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 In Fullscreen Mode ( ADD )
{
```

```
slowdown=1.0f; // Speed Up The Particles (3dfx Issue) ( ADD )
}

while(!done) // Loop That Runs Until done=TRUE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Updating View Only If Active
{
done=TRUE; // ESC or DrawGLScene Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
```

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; // Speed Up Particles
```

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+=0.01f; // Slow Down Particles
```

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 Us It's Pressed
rainbow=!rainbow; // Toggle Rainbow Mode On / Off
}
if (!keys[VK_RETURN]) rp=false; // If Return Is Released Clear Flag
```

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))) // Space Or Rainbow Mode
{
```

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 Spacebar Is Pressed Disable Rainbow Mode
```

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 Us Space Is Pressed
delay=0; // Reset The Rainbow Color Cycling Delay
col++; // Change The Particle Color
```

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 High Reset It
}
```

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 Released Clear Flag
```

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 Upward Speed
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 Downward Speed
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 Speed To The Right
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 Speed To The Left
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 Cycling Delay Counter
```

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 / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Particle Tutorial",640,480,16,fullscreen))
{
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

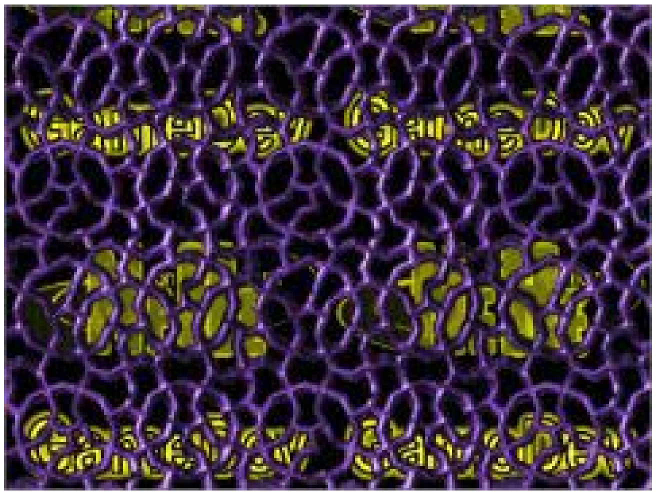
Matrix and repositioning each particle with glTranslatef(). Both methods are effective, but his method will reduce the amount of work the computer has to do before it draws each particle, causing the program to run even faster.

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**)

Lesson 20
Masking



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 Library
#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
```

We'll be using 7 global variables in this program. masking is a boolean variable (TRUE / FALSE) that will keep track of whether or not masking is turned on of off. mp is used to make sure that the 'M' key isn't being held down. sp is used to make sure that the 'Spacebar' isn't being held down and the variable scene will keep track of whether or not we're drawing the first or second scene.

We set up storage space for 5 textures using the variable texture[5]. loop is our generic counter variable, we'll use it a few times in our program to set up textures, etc. Finally we have the variable roll. We'll use roll to roll the textures across the screen. Creates a neat effect! We'll also use it to spin the object in scene 2.

```
bool keys[256]; // 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 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 Textures
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 Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator
AUX_RGBImageRec *TextureImage[5]; // Create Storage Space For The Texture Data
memset(TextureImage,0,sizeof(void *)*5); // Set The Pointer To NULL

if ((TextureImage[0]=LoadBMP("Data/logo.bmp")) && // Logo Texture
(TextureImage[1]=LoadBMP("Data/mask1.bmp")) && // First Mask
(TextureImage[2]=LoadBMP("Data/image1.bmp")) && // First Image
(TextureImage[3]=LoadBMP("Data/mask2.bmp")) && // Second Mask
(TextureImage[4]=LoadBMP("Data/image2.bmp"))) // Second Image
{
Status=TRUE; // Set The Status To TRUE
glGenTextures(5, &texture[0]); // Create Five Textures

for (loop=0; loop<5; loop++) // Loop Through All 5 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, TextureImage[loop]->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
}
}
for (loop=0; loop<5; loop++) // Loop Through All 5 Textures
{
if (TextureImage[loop]) // If Texture Exists
{
if (TextureImage[loop]->data) // If Texture Image Exists
{
free(TextureImage[loop]->data); // Free The Texture Image Memory
}
free(TextureImage[loop]); // Free The Image Structure
}
}
return Status; // Return The Status
}
```

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 Setup For OpenGL Goes Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load Return FALSE
}

glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Clear The Background Color To Black
glClearDepth(1.0); // Enables Clearing Of The Depth Buffer
glEnable(GL_DEPTH_TEST); // Enable Depth Testing
glShadeModel(GL_SMOOTH); // Enables Smooth Color Shading
glEnable(GL_TEXTURE_2D); // Enable 2D Texture Mapping
return TRUE; // Initialization Went OK
}
```

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) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The Modelview Matrix
glTranslatef(0.0f,0.0f,-2.0f); // Move Into The Screen 5 Units
```

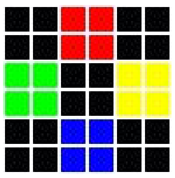
The first line below selects the 'logo' texture. We'll map the texture to the screen using a quad. We specify our four texture coordinates along with our four vertices.

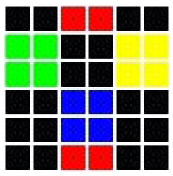
Revised Description by Jonathan Roy: Remember that OpenGL is a vertex-based graphic system. Most of the parameters you set are recorded as attributes of a particular vertex. Texture coordinate is one such attribute. You simply specify appropriate texture coordinates for each vertex of a polygon, and OpenGL automatically fills in the surface between the vertices with the texture, through a process known as interpolation. Interpolation is a standard geometric technique that lets OpenGL determine how a given parameter varies between vertices just by knowing the value that parameter takes at the vertices themselves.

Like in the previous lessons, we pretend we are facing the quad and assign texture coordinates as follows: (0.0, 0.0) to the bottom-left corner, (0.0, 1.0) to the top-left corner, (1.0, 0.0) to the bottom-right, and (1.0, 1.0) to the top-right. Now, given these settings, can you tell what texture coordinates correspond to the quad's middle point? That's right, (0.5, 0.5). But no where in the code did you specify that coordinate, did you? When it draws the quad, OpenGL computes it for you. And the real magic is that it does so whatever the position, size, or orientation of the polygon!

In this lesson we add another interesting twist by assigning texture coordinates with values other than 0.0 and 1.0. Texture coordinates are said to be normalized. Value 0.0 maps to one edge of the texture, while value 1.0 maps to the opposite edge, spanning the full width or height of the texture image in a one unit step, regardless of the polygon's size or the image's size in pixels (which we therefore don't have to worry about when doing texture mapping, and that makes life a whole lot easier). Above 1.0, the mapping simply wraps around at the other edge and the texture repeats. In other words, texture coordinate (0.3, 0.5) for instance, maps to the exact same pixel in the texture image as coordinate (1.3, 0.5), or as (12.3, -2.5). In this lesson, we achieve a tiling effect by specifying value 3.0 instead of 1.0, effectively repeating the texture nine times (3x3 tiling) over the surface of the quad.

Additionally, we use the roll variable to translate (or slide) the texture over the surface of the quad. A value of 0.0 for roll, which is added to the vertical texture coordinate, means that texture mapping on the bottom edge of the quad begins at the bottom edge of the texture image, as shown in the figure on the left. When roll equals 0.5, mapping on the bottom edge of the quad begins halfway up in the image (see figure on the right). Rolling textures can be used to create great effects such as moving clouds, words spinning around an object, etc.





```
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Our Logo
Texture
glBegin(GL_QUADS); // Start Drawing A Textured Quad
glTexCoord2f(0.0f, -roll+0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
// Bottom Left
glTexCoord2f(3.0f, -roll+0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
// Bottom Right
glTexCoord2f(3.0f, -roll+3.0f); glVertex3f( 1.1f, 1.1f, 0.0f); // Top Right
glTexCoord2f(0.0f, -roll+3.0f); glVertex3f(-1.1f, 1.1f, 0.0f); // Top Left
glEnd(); // Done Drawing The Quad
```

Anyways... back to reality. Now we enable blending. In order for this effect to work we also have to disable depth testing. It's very important that you do this! If you do not disable depth testing you probably wont see anything. Your entire image will vanish!

```
glEnable(GL_BLEND); // Enable Blending
glDisable(GL_DEPTH_TEST); // Disable Depth Testing
```

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) // Is Masking Enabled?
{
```

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 Screen Color With Zero (Black)
}
```

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) // Are We Drawing The Second 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); // Translate Into The Screen One Unit
glRotatef(roll*360,0.0f,0.0f,1.0f); // Rotate On The Z Axis 360 Degrees
```

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) // Is Masking On?
{
```

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 The Second Mask Texture
glBegin(GL_QUADS); // Start Drawing A Textured Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f); // Bottom Left
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f); // Bottom Right
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f, 1.1f, 0.0f); // Top Right
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f, 1.1f, 0.0f); // Top Left
glEnd(); // Done Drawing The Quad
}
```

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); // Copy Image 2 Color To The Screen
glBindTexture(GL_TEXTURE_2D, texture[4]); // Select The Second Image Texture
glBegin(GL_QUADS); // Start Drawing A Textured Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f); // Bottom Left
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f); // Bottom Right
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f, 1.1f, 0.0f); // Top Right
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f, 1.1f, 0.0f); // Top Left
glEnd(); // Done Drawing The Quad
}
```

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

```
else // Otherwise
{
```

We start off by checking to see if masking is TRUE of FALSE, just like in the code above.

```
if (masking) // Is Masking On?
{
```

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 The First Mask Texture
glBegin(GL_QUADS); // Start Drawing A Textured Quad
glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f); // Bottom Left
glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f); // Bottom Right
glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f, 1.1f, 0.0f); // Top Right
glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f, 1.1f, 0.0f); // Top Left
glEnd(); // Done Drawing The Quad
}
```

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); // Copy Image 1 Color To The Screen
glBindTexture(GL_TEXTURE_2D, texture[2]); // Select The First Image Texture
glBegin(GL_QUADS); // Start Drawing A Textured Quad
glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f); // Bottom Left
glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f); // Bottom Right
glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f, 1.1f, 0.0f); // Top Right
glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f, 1.1f, 0.0f); // Top Left
glEnd(); // Done Drawing The Quad
}
```

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 Depth Testing
glDisable(GL_BLEND); // Disable Blending
```

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; // Increase Our Texture Roll Variable
if (roll>1.0f) // Is Roll Greater Than One
{
roll-=1.0f; // Subtract 1 From Roll
}

return TRUE; // Everything Went OK
}
```

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 hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}

// Create Our OpenGL Window
if (!CreateGLWindow("NeHe's Masking Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit Received?
{
done=TRUE; // ESC or DrawGLScene Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
```

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) // Is Space Being Pressed?
{
sp=TRUE; // Tell Program Spacebar Is Being Held
scene=!scene; // Toggle From One Scene To The Other
}
```

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[' ']) // Has Spacebar Been Released?
{
sp=FALSE; // Tell Program Spacebar Has Been Released
}
```

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) // Is M Being Pressed?
{
mp=TRUE; // Tell Program M Is Being Held
masking=!masking; // Toggle Masking Mode OFF/ON
}
```

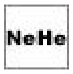
The last bit of code checks to see if we've stopped pressing 'M'. If we have, mp becomes FALSE letting the program know that we are no longer holding the 'M' key down. Once the 'M' key has been released, we are able to press it once again to toggle masking on or off.

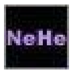
```
if (!keys['M']) // Has M Been Released?
{
mp=FALSE; // Tell Program That M Has Been Released
}
```

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 / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Masking Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
}
}
}
// Shutdown
KillGLWindow(); // Kill The Window
return (msg.wParam); // Exit The Program
}
```

Creating a mask isn't to hard. A little time consuming. The best way to make a mask if you already have your image made is to load your image into an art program or a handy program like infranview, and reduce it to a gray scale image. After you've done that, turn the contrast way up so that gray pixels become black. You can also try turning down the brightness, etc. It's important that the white is bright white, and the black is pure black. If you have any gray pixels in your mask, that section of the image will appear transparent. The most reliable way to make sure your mask is a perfect copy of your image is to trace over the image with black. It's also very important that your image has a BLACK background and the mask has a WHITE background! If you create a mask and notice a square shape around your texture, either your white isn't bright enough (255 or FFFFFF) or your black isn't true black (0 or 000000). Below you can see an example of a mask and the image that goes over top of the mask. the image can be any color you want as long as the background is black. The mask must have a white background and a black copy of your image.

This is the mask -> 

This is the image -> 

Eric Desrosiers pointed out that you can also check the value of each pixel in your bitmap while you load it. If you want the pixel transparent you can give it an alpha value of 0. For all the other colors you can give them an alpha value of 255. This method will also work but requires some extra coding. The current tutorial is simple and requires very little extra code. I'm not blind to other techniques, but when I write a tutorial I try to make the code easy to understand and easy to use. I just wanted to point out that there are always other ways to get the job done. Thanks for the feedback Eric.

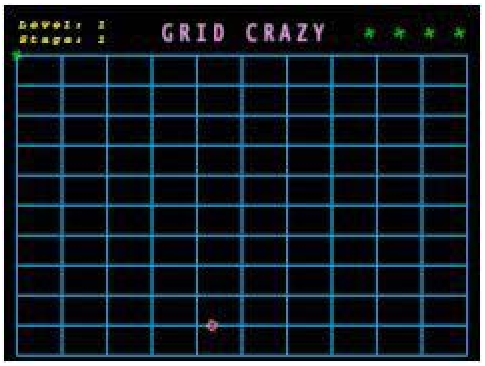
In this tutorial I have shown you a simple, but effective way to draw sections of a texture to the screen without using the alpha channel. Normal blending usually looks bad (textures are either transparent or they're not), and texturing with an alpha channel requires that your images support the alpha channel. Bitmaps are convenient to work with, but they do not support the alpha channel this program shows us how to get around the limitations of bitmap images, while demonstrating a cool way to create overlay type effects.

Thanks to Rob Santa for the idea and for example code. I had never heard of this little trick until he pointed it out. He wanted me to point out that although this trick does work, it takes two passes, which causes a performance hit. He recommends that you use textures that support the alpha channel for complex scenes.

I hope you enjoyed this tutorial. If you had any problems understanding it, or you've found a mistake in the tutorial please let me know. I want to make the best tutorials available. Your feedback is important!

Jeff Molofee (**NeHe**)

Lesson 21
Lines, Antialiasing, Timing, Ortho View And Simple Sounds



Welcome to my 21st OpenGL Tutorial! Coming up with a topic for this tutorial was extremely difficult. I know alot of you are tired of learning the basics. Everyone is dying to learn about 3D objects, Multitexturing and all that other good stuff. For those people, I'm sorry, but I want to keep the learning curve gradual. Once I've gone a step ahead it's not as easy to take a step back without people losing interest. So I'd prefer to keep pushing forward at a steady pace.

In case I've lost a few of you :) I'll tell you a bit about this tutorial. Until now all of my tutorials have used polygons, quads and triangles. So I decided it would be nice to write a tutorial on lines. A few hours after starting the line tutorial, I decided to call it quits. The tutorial was coming along fine, but it was BORING! Lines are great, but there's only so much you can do to make lines exciting. I read through my email, browsed through the message board, and wrote down a few of your tutorial requests. Out of all the requests there were a few questions that came up more than others. So... I decided to write a multi-tutorial :)

In this tutorial you will learn about: Lines, Anti-Aliasing, Orthographic Projection, Timing, Basic Sound Effects, and Simple Game Logic. Hopefully there's enough in this tutorial to keep everyone happy :) I spent 2 days coding this tutorial, and It's taken almost 2 weeks to write this HTML file. I hope you enjoy my efforts!

At the end of this tutorial you will have made a simple 'amidar' type game. Your mission is to fill in the grid without being caught by the bad guys. The game has levels, stages, lives, sound, and a secret item to help you progress through the levels when things get tough. Although this game will run fine on a Pentium 166 with a Voodoo 2, a faster processor is recommended if you want smoother animation.

I used the code from lesson 1 as a starting point while writing this tutorial. We start off by adding the required header files. stdio.h is used for file operations, and we include stdarg.h so that we can display variables on the screen, such as the score and current stage.

```
// This Code Was Created By Jeff Molofee 2000
// If You've Found This Code Useful, Please Let Me Know.

#include <windows.h> // Header File For Windows
#include <stdio.h> // Standard Input / Output
#include <stdarg.h> // Header File For Variable Argument Routines
#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
```

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]; // Array Used For The Keyboard Routine
bool vline[11][10]; // Keeps Track Of Verticle Lines
bool hline[10][11]; // Keeps Track Of Horizontal Lines
bool ap; // 'A' Key Pressed?
bool filled; // Done Filling In The Grid?
bool gameover; // Is The Game Over?
bool anti=TRUE; // Antialiasing?
bool active=TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
```

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; // Generic Loop1
int loop2; // Generic Loop2
int delay; // Enemy Delay
int adjust=3; // Speed Adjustment For Really Slow Video Cards
int lives=5; // Player Lives
int level=1; // Internal Game Level
int level2=level; // Displayed Game Level
int stage=1; // Game Stage
```

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 // Create A Structure For Our Player
{
int fx, fy; // Fine Movement Position
int x, y; // Current Player Position
float spin; // Spin Direction
};
```

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; // Player Information
struct object enemy[9]; // Enemy Information
struct object hourglass; // Hourglass Information
```

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 // Create A Structure For The Timer Information
{
__int64 frequency; // Timer Frequency
float resolution; // Timer Resolution
unsigned long mm_timer_start; // Multimedia Timer Start Value
unsigned long mm_timer_elapsed; // Multimedia Timer Elapsed Time
bool performance_timer; // Using The Performance Timer?
__int64 performance_timer_start; // Performance Timer Start Value
__int64 performance_timer_elapsed; // Performance Timer Elapsed Time
} timer; // Structure Is Named timer
```

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 }; // Stepping Values For Slow Video Adjustment
```

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]; // Font Texture Storage Space
GLuint base; // Base Display List For The Font

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

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) // Initialize Our Timer (Get It Ready)
{
memset(&timer, 0, sizeof(timer)); // Clear Our Timer Structure

// Check To See If A Performance Counter Is Available
// If One Is Available The Timer Frequency Will Be Updated
if (!QueryPerformanceFrequency((LARGE_INTEGER *) &timer.frequency))
{
// No Performace Counter Available
timer.performance_timer = FALSE; // Set Performance Timer To FALSE
timer.mm_timer_start = timeGetTime(); // Use timeGetTime() To Get Current Time
timer.resolution = 1.0f/1000.0f; // Set Our Timer Resolution To .001f
timer.frequency = 1000; // Set Our Timer Frequency To 1000
timer.mm_timer_elapsed = timer.mm_timer_start; // Set The Elapsed Time To The Current Time
}
```

If there is a performance counter, the following code is run instead. The first line grabs the current starting value of the performance counter, and stores it in timer.performance_timer_start. Then we set timer.performance_timer to TRUE so that our program knows there is a performance counter available. After that we calculate the timer resolution by using the frequency that we got when we checked for a performance counter in the code above. We divide 1 by the frequency to get the resolution. The last thing we do is make the elapsed time the same as the starting time.

Notice instead of sharing variables for the performance and multimedia timer start and elapsed variables, I've decided to make seperate variables. Either way it will work fine.

```
else
{
// Performance Counter Is Available, Use It Instead Of The Multimedia Timer
// Get The Current Time And Store It In performance_timer_start
QueryPerformanceCounter((LARGE_INTEGER *) &timer.performance_timer_start);
timer.performance_timer = TRUE; // Set Performance Timer To TRUE
// Calculate The Timer Resolution Using The Timer Frequency
timer.resolution = (float) (((double)1.0f)/((double)timer.frequency));
// Set The Elapsed Time To The Current Time
timer.performance_timer_elapsed = timer.performance_timer_start;
}
}
```

The section of code above sets up the timer. The code below reads the timer and returns the amount of time that has passed in milliseconds.

The first thing we do is set up a 64 bit variable called time. We will use this variable to grab the current counter value. The next line checks to see if we have a performance counter. If we do, timer.performance_timer will be TRUE and the code right after will run.

The first line of code inside the { }'s grabs the counter value and stores it in the variable we created called time. The second line takes the time we just grabbed (time and subtracts the start time that we got when we initialized the timer. This way our timer should start out pretty close to zero. We then multiply the results by the resolution to find out how many seconds have passed. The last thing we do is multiply the result by 1000 to figure out how many milliseconds have passed. After the calculation is done, our results are sent back to the section of code that called this procedure. The results will be in floating point format for greater accuracy.

If we are not using the peformance counter, the code after the else statement will be run. It does pretty much the same thing. We grab the current time with timeGetTime() and subtract our starting counter value. We multiply it by our resolution and then multiply the result by 1000 to convert from seconds into milliseconds.

```
float TimerGetTime() // Get Time In Milliseconds
{
__int64 time; // time Will Hold A 64 Bit Integer

if (timer.performance_timer) // Are We Using The Performance Timer?
{
QueryPerformanceCounter((LARGE_INTEGER *) &time); // Grab The Current Performance Time
// Return The Current Time Minus The Start Time Multiplied By The Resolution And 1000 (To Get
MS)
return ( (float) ( time - timer.performance_timer_start) * timer.resolution)*1000.0f;
}
else
{
// Return The Current Time Minus The Start Time Multiplied By The Resolution And 1000 (To Get
MS)
return( (float) ( timeGetTime() - timer.mm_timer_start) * timer.resolution)*1000.0f;
}
}
```

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) // Reset Player And Enemies
{
player.x=0; // Reset Player X Position To Far Left Of The Screen
player.y=0; // Reset Player Y Position To The Top Of The Screen
player.fx=0; // Set Fine X Position To Match
player.fy=0; // Set Fine Y Position To Match
```

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 Through All The Enemies
{
enemy[loop1].x=5+rand()%6; // Select A Random X Position
enemy[loop1].y=rand()%11; // Select A Random Y Position
enemy[loop1].fx=enemy[loop1].x*60; // Set Fine X To Match
enemy[loop1].fy=enemy[loop1].y*40; // Set Fine Y To Match
}
}
```

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() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator
AUX_RGBImageRec *TextureImage[2]; // Create Storage Space For The Textures
memset(TextureImage,0,sizeof(void *)*2); // Set The Pointer To NULL

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

glGenTextures(2, &texture[0]); // Create The Texture

for (loop1=0; loop1<2; loop1++) // Loop Through 2 Textures
{
glBindTexture(GL_TEXTURE_2D, texture[loop1]);
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop1]->sizeX, TextureImage[loop1]->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop1]->data);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
}

for (loop1=0; loop1<2; loop1++) // Loop Through 2 Textures
{
if (TextureImage[loop1]) // If Texture Exists
{
if (TextureImage[loop1]->data) // If Texture Image Exists
{
free(TextureImage[loop1]->data); // Free The Texture Image Memory
}
free(TextureImage[loop1]); // Free The Image Structure
}
}
}
return Status; // Return The 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 Our Font Display List
{
base=glGenLists(256); // Creating 256 Display Lists
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Our Font Texture
for (loop1=0; loop1<256; loop1++) // Loop Through All 256 Lists
{
float cx=float(loop1%16)/16.0f; // X Position Of Current Character
float cy=float(loop1/16)/16.0f; // Y Position Of Current Character

glNewList(base+loop1,GL_COMPILE); // Start Building A List
glBegin(GL_QUADS); // Use A Quad For Each Character
glTexCoord2f(cx,1.0f-cy-0.0625f); // Texture Coord (Bottom Left)
glVertex2d(0,16); // Vertex Coord (Bottom Left)
glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f); // Texture Coord (Bottom Right)
glVertex2i(16,16); // Vertex Coord (Bottom Right)
glTexCoord2f(cx+0.0625f,1.0f-cy); // Texture Coord (Top Right)
glVertex2i(16,0); // Vertex Coord (Top Right)
glTexCoord2f(cx,1.0f-cy); // Texture Coord (Top Left)
glVertex2i(0,0); // Vertex Coord (Top Left)
glEnd(); // Done Building Our Quad (Character)
glTranslated(15,0,0); // Move To The Right Of The Character
glEndList(); // Done Building The Display List
} // Loop Until All 256 Are Built
}
```

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) // Delete The Font From Memory
{
glDeleteLists(base,256); // Delete All 256 Display Lists
}
```

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 The Printing Happens
{
char text[256]; // Holds Our String
va_list ap; // Pointer To List Of Arguments

if (fmt == NULL) // If There's No Text
return; // Do Nothing

va_start(ap, fmt); // Parses The String For Variables
vsprintf(text, fmt, ap); // And Converts Symbols To Actual Numbers
va_end(ap); // Results Are Stored In Text

if (set>1) // Did User Choose An Invalid Character Set?
{
set=1; // If So, Select Set 1 (Italic)
}
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glLoadIdentity(); // Reset The Modelview Matrix
glTranslated(x,y,0); // Position The Text (0,0 - Bottom Left)
glListBase(base-32+(128*set)); // Choose The Font Set (0 or 1)

if (set==0) // If Set 0 Is Being Used Enlarge Font
{
glScalef(1.5f,2.0f,1.0f); // Enlarge Font Width And Height
}

glCallLists(strlen(text),GL_UNSIGNED_BYTE, text); // Write The Text To The Screen
glDisable(GL_TEXTURE_2D); // Disable Texture Mapping
}
```

The resize code is NEW :) Instead of using a perspective view I'm using an ortho view for this tutorial. That means that objects don't get smaller as they move away from the viewer. The z-axis is pretty much useless in this tutorial.

We start off by setting up the view port. We do this the same way we'd do it if we were setting up a perspective view. We make the viewport equal to the width of our window.

Then we select the projection matrix (thing movie projector, it information on how to display our image). and reset it.

Immediately after we reset the projection matrix, we set up our ortho view. I'll explain the command in detail:

The first parameter (0.0f) is the value that we want for the far left side of the screen. You wanted to know how to use actual pixel values, so instead of using a negative number for far left, I've set the value to 0. The second parameter is the value for the far right side of the screen. If our window is 640x480, the value stored in width will be 640. So the far right side of the screen effectively becomes 640. Therefore our screen runs from 0 to 640 on the x-axis.

The third parameter (height) would normally be our negative y-axis value (bottom of the screen). But because we want exact pixels, we wont have a negative value. Instead we will make the bottom of the screen equal the height of our window. If our window is 640x480, height will be equal to 480. So the bottom of our screen will be 480. The fourth parameter would normally be the positive value for the top of our screen. We want the top of the screen to be 0 (good old fashioned screen coordinates) so we just set the fourth parameter to 0. This gives us from 0 to 480 on the y-axis.

The last two parameters are for the z-axis. We don't really care about the z-axis so we'll set the range from -1.0f to 1.0f. Just enough that we can see anything drawn at 0.0f on the z-axis.

After we've set up the ortho view, we select the modelview matrix (object information... location, etc) and reset it.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Window
{
if (height==0) // Prevent A Divide By Zero By
{
height=1; // Making Height Equal One
}

glViewport(0,0,width,height); // Reset The Current Viewport

glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
glLoadIdentity(); // Reset The Projection Matrix

glOrtho(0.0f,width,height,0.0f,-1.0f,1.0f); // Create Ortho 640x480 View (0,0 At Top Left)

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

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

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
if (!LoadGLTextures()) // Jump To Texture Loading Routine
{
return FALSE; // If Texture Didn't Load Return FALSE
}

BuildFont(); // Build The Font

glShadeModel(GL_SMOOTH); // Enable Smooth Shading
glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Black Background
glClearDepth(1.0f); // Depth Buffer Setup
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST); // Set Line Antialiasing
glEnable(GL_BLEND); // Enable Blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Type Of Blending To Use
return TRUE; // Initialization Went OK
}
```

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) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Our Font Texture
glColor3f(1.0f,0.5f,1.0f); // Set Color To Purple
glPrint(207,24,0,"GRID CRAZY"); // Write GRID CRAZY On The Screen
glColor3f(1.0f,1.0f,0.0f); // Set Color To Yellow
glPrint(20,20,1,"Level:%2i",level2); // Write Actual Level Stats
glPrint(20,40,1,"Stage:%2i",stage); // Write Stage Stats
```

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) // Is The Game Over?
{
glColor3ub(rand()%255,rand()%255,rand()%255); // Pick A Random Color
glPrint(472,20,1,"GAME OVER"); // Write GAME OVER To The Screen
glPrint(456,40,1,"PRESS SPACE"); // Write PRESS SPACE To The Screen
}
```

If the player still has lives left, we draw animated images of the players character to the right of the game title. To do this we create a loop that goes from 0 to the current number of lives the player has left minus one. I subtract one, because the current life is the image you control.

Inside the loop, we reset the view. After the view has been reset, we translate to the 490 pixels to the right plus the value of loop1 times 40.0f. This draws each of the animated player lives 40 pixels apart from eachother. The first animated image will be drawn at 490+(0*40) (= 490), the second animated image will be drawn at 490+(1*40) (= 530), etc.

After we have moved to the spot we want to draw the animated image, we rotate counterclockwise depending on the value stored in player.spin. This causes the animated life images to spin the opposite way that your active player is spinning.

We then select green as our color, and start drawing the image. Drawing lines is alot like drawing a quad or a polygon. You start off with glBegin(GL_LINES), telling OpenGL we want to draw a line. Lines have 2 vertices. We use glVertex2d to set our first point. glVertex2d doesn't require a z value, which is nice considering we don't care about the z value. The first point is drawn 5 pixels to the left of the current x location and 5 pixels up from the current y location. Giving us a top left point. The second point of our first line is drawn 5 pixels to the right of our current x location, and 5 pixels down, giving us a bottom right point. This draws a line from the top left to the bottom right. Our second line is drawn from the top right to the bottom left. This draws a green X on the screen.

After we have drawn the green X, we rotate counterclockwise (on the z axis) even more, but this time at half the speed. We then select a darker shade of green (0.75f) and draw another x, but we use 7 instead of 5 this time. This draws a bigger / darker x on top of the first green X. Because the darker X spins slower though, it will look as if the bright X has a spinning set of feelers (grin) on top of it.

```
for (loop1=0; loop1<lives-1; loop1++) // Loop Through Lives Minus Current Life
{
glLoadIdentity(); // Reset The View
glTranslatef(490+(loop1*40.0f),40.0f,0.0f); // Move To The Right Of Our Title Text
glRotatef(-player.spin,0.0f,0.0f,1.0f); // Rotate Counter Clockwise
glColor3f(0.0f,1.0f,0.0f); // Set Player Color To Light Green
glBegin(GL_LINES); // Start Drawing Our Player Using Lines
glVertex2d(-5,-5); // Top Left Of Player
glVertex2d( 5, 5); // Bottom Right Of Player
glVertex2d( 5,-5); // Top Right Of Player
glVertex2d(-5, 5); // Bottom Left Of Player
glEnd(); // Done Drawing The Player
glRotatef(-player.spin*0.5f,0.0f,0.0f,1.0f); // Rotate Counter Clockwise
glColor3f(0.0f,0.75f,0.0f); // Set Player Color To Dark Green
glBegin(GL_LINES); // Start Drawing Our Player Using Lines
glVertex2d(-7, 0); // Left Center Of Player
glVertex2d( 7, 0); // Right Center Of Player
glVertex2d( 0,-7); // Top Center Of Player
glVertex2d( 0, 7); // Bottom Center Of Player
glEnd(); // Done Drawing The Player
}
```

Now we're going to draw the grid. We set the variable filled to TRUE. This tells our program that the grid has been completely filled in (you'll see why we do this in a second).

Right after that we set the line width to 2.0f. This makes the lines thicker, making the grid look more defined.

Then we disable anti-aliasing. The reason we disable anti-aliasing is because although it's a great feature, it eats CPU's for breakfast. Unless you have a killer graphics card, you'll notice a huge slow down if you leave anti-aliasing on. Go ahead and try if you want :)

The view is reset, and we start two loops. loop1 will travel from left to right. loop2 will travel from top to bottom.

We set the line color to blue, then we check to see if the horizontal line that we are about to draw has been traced over. If it has we set the color to white. The value of hline[loop1][loop2] will be TRUE if the line has been traced over, and FALSE if it hasn't.

After we have set the color to blue or white, we draw the line. The first thing to do is make sure we haven't gone to far to the right. We don't want to draw any lines or check to see if the line has been filled in when loop1 is greater than 9.

Once we are sure loop1 is in the valid range we check to see if the horizontal line hasn't been filled in. If it hasn't, filled is set to FALSE, letting our OpenGL program know that there is at least one line that hasn't been filled in.

The line is then drawn. We draw our first horizontal (left to right) line starting at 20+(0*60) (= 20). This line is drawn all the way to 80+(0*60) (= 80). Notice the line is drawn to the right. That is why we don't want to draw 11 (0-10) lines. because the last line would start at the far right of the screen and end 80 pixels off the screen.

```
filled=TRUE; // Set Filled To True Before Testing
glLineWidth(2.0f); // Set Line Width For Cells To 2.0f
glDisable(GL_LINE_SMOOTH); // Disable Antialiasing
glLoadIdentity(); // Reset The Current Modelview Matrix
for (loop1=0; loop1<11; loop1++) // Loop From Left To Right
{
for (loop2=0; loop2<11; loop2++) // Loop From Top To Bottom
{
glColor3f(0.0f,0.5f,1.0f); // Set Line Color To Blue
if (hline[loop1][loop2]) // Has The Horizontal Line Been Traced
{
glColor3f(1.0f,1.0f,1.0f); // If So, Set Line Color To White
}
if (loop1<10) // Dont Draw To Far Right
{
if (!hline[loop1][loop2]) // If A Horizontal Line Isn't Filled
{
filled=FALSE; // filled Becomes False
}
glBegin(GL_LINES); // Start Drawing Horizontal Cell Borders
glVertex2d(20+(loop1*60),70+(loop2*40)); // Left Side Of Horizontal Line
glVertex2d(80+(loop1*60),70+(loop2*40)); // Right Side Of Horizontal Line
glEnd(); // Done Drawing Horizontal Cell Borders
}
```

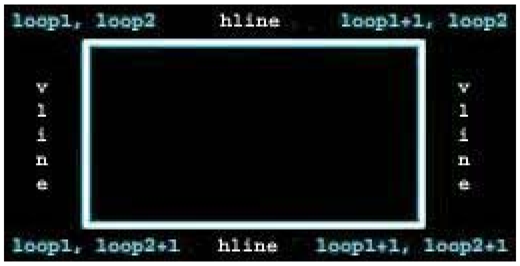
The code below does the same thing, but it checks to make sure the line isn't being drawn too far down the screen instead of too far right. This code is responsible for drawing vertical lines.

```
glColor3f(0.0f,0.5f,1.0f); // Set Line Color To Blue
if (vline[loop1][loop2]) // Has The Horizontal Line Been Traced
{
glColor3f(1.0f,1.0f,1.0f); // If So, Set Line Color To White
}
if (loop2<10) // Dont Draw To Far Down
{
if (!vline[loop1][loop2]) // If A Verticle Line Isn't Filled
{
filled=FALSE; // filled Becomes False
}
glBegin(GL_LINES); // Start Drawing Verticle Cell Borders
glVertex2d(20+(loop1*60),70+(loop2*40)); // Left Side Of Horizontal Line
glVertex2d(20+(loop1*60),110+(loop2*40)); // Right Side Of Horizontal Line
glEnd(); // Done Drawing Verticle Cell Borders
}
```

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 runs along loop2. The second horizontal line runs along loop2+1. Vertical lines are assumed to run from loop2 to loop2+1. The first vertical line runs along loop1 and the second vertical line runs along loop1+1

When loop1 is increased, the right side of our old box becomes the left side of the new box. When loop2 is increased, the bottom of the old box becomes the top of the new box.

If all 4 borders are TRUE (meaning we've passed over them all) we can texture map the box. We do this the same way we broke the font texture into seperate letters. We divide both loop1 and loop2 by 10 because we want to map the texture across 10 boxes from left to right and 10 boxes up and down. Texture coordinates run from 0.0f to 1.0f and 1/10th of 1.0f is 0.1f.

So to get the top right side of our box we divide the loop values by 10 and add 0.1f to the x texture coordinate. To get the top left side of the box we divide our loop values by 10. To get the bottom left side of the box we divide our loop values by 10 and add 0.1f to the y texture coordinate. Finally to get the bottom right texture coordinate we divide the loop values by 10 and add 0.1f to both the x and y texture coordinates.

Quick examples: loop1=0 and loop2=0

- Right X Texture Coordinate = loop1/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f
- Left X Texture Coordinate = loop1/10 = 0/10 = 0.0f
- Top Y Texture Coordinate = loop2/10 = 0/10 = 0.0f;
- Bottom Y Texture Coordinate = loop2/10+0.1f = 0/10+0.1f = 0+0.1f = 0.1f;

loop1=1 and loop2=1

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

Hopefully that all makes sense. If loop1 and loop2 were equal to 9 we would end up with the values 0.9f and 1.0f. So as you can see our texture coordinates mapped across the 10 boxes run from 0.0f at the lowest and 1.0f at the highest. Mapping the entire texture to the screen. After we've mapped a section of the texture to the screen, we disable texture mapping. Once we've drawn all the lines and filled in all the boxes, we set the line width to 1.0f.

```
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glColor3f(1.0f,1.0f,1.0f); // Bright White Color
glBindTexture(GL_TEXTURE_2D, texture[1]); // Select The Tile Image
if ((loop1<10) && (loop2<10)) // If In Bounds, Fill In Traced Boxes
{
// Are All Sides Of The Box Traced?
if (hline[loop1][loop2] && hline[loop1][loop2+1] && vline[loop1][loop2] &&
vline[loop1+1][loop2])
{
glBegin(GL_QUADS); // Draw A Textured Quad
glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float(loop2/10.0f)));
glVertex2d(20+(loop1*60)+59,(70+loop2*40+1)); // Top Right
glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)));
glVertex2d(20+(loop1*60)+1,(70+loop2*40+1)); // Top Left
glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)+0.1f));
glVertex2d(20+(loop1*60)+1,(70+loop2*40)+39); // Bottom Left
glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float(loop2/10.0f)+0.1f));
glVertex2d(20+(loop1*60)+59,(70+loop2*40)+39); // Bottom Right
glEnd(); // Done Texturing The Box
}
}
glDisable(GL_TEXTURE_2D); // Disable Texture Mapping
}
}
glLineWidth(1.0f); // Set The Line Width To 1.0f
```

The code below checks to see if anti is TRUE. If it is, we enable line smoothing (anti-aliasing).

```
if (anti) // Is Anti TRUE?
{
glEnable(GL_LINE_SMOOTH); // If So, Enable Antialiasing
}
```

To make the game a little easier I've added a special item. The item is an hourglass. When you touch the hourglass, the enemies are frozen for a specific amount of time. The following section of code is resposible for drawing the hourglass.

For the hourglass we use x and y to position the timer, but unlike our player and enemies we don't use fx and fy for fine positioning. Instead we'll use fx to keep track of whether or not the timer is being displayed. fx will equal 0 if the timer is not visible. 1 if it is visible, and 2 if the player has touched the timer. fy will be used as a counter to keep track of how long the timer should be visible or invisible.

So we start off by checking to see if the timer is visible. If not, we skip over the code without drawing the timer. If the timer is visible, we reset the modelview matrix, and position the timer. Because our first grid point from left to right starts at 20, we will add hourglass.x times 60 to 20. We multiply hourglass.x by 60 because the points on our grid from left to right are spaced 60 pixels apart. We then position the hourglass on the y axis. We add hourglass.y times 40 to 70.0f because we want to start drawing 70 pixels down from the top of the screen. Each point on our grid from top to bottom is spaced 40 pixels apart.

After we have positioned the hourglass, we can rotate it on the z-axis. hourglass.spin is used to keep track of the rotation, the same way player.spin keeps track of the player rotation. Before we start to draw the hourglass we select a random color.

```
if (hourglass.fx==1) // If fx=1 Draw The Hourglass
{
```

```
glLoadIdentity(); // Reset The Modelview Matrix
glTranslatef(20.0f+(hourglass.x*60),70.0f+(hourglass.y*40),0.0f); // Move To The Fine Hourglass
Position
glRotatef(hourglass.spin,0.0f,0.0f,1.0f); // Rotate Clockwise
glColor3ub(rand()%255,rand()%255,rand()%255); // Set Hourglass Color To Random Color
```

glBegin(GL_LINES) tells OpenGL we want to draw using lines. We start off by moving left and up 5 pixels from our current location. This gives us the top left point of our hourglass. OpenGL will start drawing the line from this location. The end of the line will be 5 pixels right and down from our original location. This gives us a line running from the top left to the bottom right. Immediately after that we draw a second line running from the top right to the bottom left. This gives us an 'X'. We finish off by connecting the bottom two points together, and then the top two points to create an hourglass type object :)

```
glBegin(GL_LINES); // Start Drawing Our Hourglass Using Lines
glVertex2d(-5,-5); // Top Left Of Hourglass
glVertex2d( 5, 5); // Bottom Right Of Hourglass
glVertex2d( 5,-5); // Top Right Of Hourglass
glVertex2d(-5, 5); // Bottom Left Of Hourglass
glVertex2d(-5, 5); // Bottom Left Of Hourglass
glVertex2d( 5, 5); // Bottom Right Of Hourglass
glVertex2d(-5,-5); // Top Left Of Hourglass
glVertex2d( 5,-5); // Top Right Of Hourglass
glEnd(); // Done Drawing The Hourglass
}
```

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 The Modelview Matrix
glTranslatef(player.fx+20.0f,player.fy+70.0f,0.0f); // Move To The Fine Player Position
glRotatef(player.spin,0.0f,0.0f,1.0f); // Rotate Clockwise
glColor3f(0.0f,1.0f,0.0f); // Set Player Color To Light Green
glBegin(GL_LINES); // Start Drawing Our Player Using Lines
glVertex2d(-5,-5); // Top Left Of Player
glVertex2d( 5, 5); // Bottom Right Of Player
glVertex2d( 5,-5); // Top Right Of Player
glVertex2d(-5, 5); // Bottom Left Of Player
glEnd(); // Done Drawing The Player
```

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 Clockwise
glColor3f(0.0f,0.75f,0.0f); // Set Player Color To Dark Green
glBegin(GL_LINES); // Start Drawing Our Player Using Lines
glVertex2d(-7, 0); // Left Center Of Player
glVertex2d( 7, 0); // Right Center Of Player
glVertex2d( 0,-7); // Top Center Of Player
glVertex2d( 0, 7); // Bottom Center Of Player
glEnd(); // Done Drawing The Player
```

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 To Draw Enemies
{
glLoadIdentity(); // Reset The Modelview Matrix
glTranslatef(enemy[loop1].fx+20.0f,enemy[loop1].fy+70.0f,0.0f);
glColor3f(1.0f,0.5f,0.5f); // Make Enemy Body Pink
glBegin(GL_LINES); // Start Drawing Enemy
glVertex2d( 0,-7); // Top Point Of Body
glVertex2d(-7, 0); // Left Point Of Body
glVertex2d(-7, 0); // Left Point Of Body
```

```
glVertex2d( 0, 7); // Bottom Point Of Body
glVertex2d( 0, 7); // Bottom Point Of Body
glVertex2d( 7, 0); // Right Point Of Body
glVertex2d( 7, 0); // Right Point Of Body
glVertex2d( 0,-7); // Top Point Of Body
glEnd(); // Done Drawing Enemy Body
```

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 The Enemy Blade
glColor3f(1.0f,0.0f,0.0f); // Make Enemy Blade Red
glBegin(GL_LINES); // Start Drawing Enemy Blade
glVertex2d(-7,-7); // Top Left Of Enemy
glVertex2d( 7, 7); // Bottom Right Of Enemy
glVertex2d(-7, 7); // Bottom Left Of Enemy
glVertex2d( 7,-7); // Top Right Of Enemy
glEnd(); // Done Drawing Enemy Blade
}
return TRUE; // Everything Went OK
}
```

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) // Properly Kill The Window
{
if (fullscreen) // Are We In Fullscreen Mode?
{
ChangeDisplaySettings(NULL,0); // If So Switch Back To The Desktop
ShowCursor(TRUE); // Show Mouse Pointer
}

if (hRC) // Do We Have A Rendering Context?
{
if (!wglMakeCurrent(NULL,NULL)) // Are We Able To Release The DC And RC Contexts?
{
MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
}

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

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

if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?
{
MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hWnd=NULL; // Set hWnd To NULL
}

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

KillFont(); // Kill The Font We Built
}
```

The CreateGLWindow() and WndProc() code hasn't changed so search until you find the following section of code.

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

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

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. TimerInit() initializes the timer so it's set up properly.

```
if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen)) // Create Our OpenGL Window
{
return 0; // Quit If Window Was Not Created
}

ResetObjects(); // Set Player / Enemy Starting Positions
TimerInit(); // Initialize The Timer

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
```

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 Timer Value Before We Draw

// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit Received?
{
done=TRUE; // ESC or DrawGLScene Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
}

while(TimerGetTime()<start+float(steps[adjust]*2.0f)) {}// Waste Cycles On Fast Systems
```

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 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
```

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) // If 'A' Key Is Pressed And Not Held
{
ap=TRUE; // ap Becomes TRUE
anti=!anti; // Toggle Antialiasing
}
if (!keys['A']) // If 'A' Key Has Been Released
{
ap=FALSE; // ap Becomes FALSE
}
```

Now to move the enemies. I wanted to keep this section of code really simple. There is very little logic. Basically, the enemies check to see where you are and they move in that direction. Because I'm checking the actual x and y position of the players and no the fine values, the players seem to have a little more intelligence. They may see that you are way at the top of the screen. But by the time they're fine value actually gets to the top of the screen, you could already be in a different location. This causes them to sometimes move past you, before they realize you are no longer where they thought you were. May sound like they're really dumb, but because they sometimes move past you, you might find yourself being boxed in from all directions.

We start off by checking to make sure the game isn't over, and that the window (if in windowed mode) is still active. By checking active the enemies wont move if the screen is minimized. This gives you a convenient pause feature when you need to take a break :)

After we've made sure the enemies should be moving, we create a loop. The loop will loop through all the visible enemies. Again we calculate how many enemies should be on the screen by multiplying the current stage by the current internal level.

```
if (!gameover && active) // If Game Isn't Over And Programs Active Move Objects
{
for (loop1=0; loop1<(stage*level); loop1++) // Loop Through The Different Stages
{
```

Now we move the current enemy (enemy[loop1]). We start off by checking to see if the enemy's x position is less than the players x position and we make sure that the enemy's fine y position lines up with a horizontal line. We can't move the enemy left and right if it's not on a horizontal line. If we did, the enemy would cut right through the middle of the boxes, making the game even more difficult :)

If the enemy 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].fy==enemy[loop1].y*40))
{
enemy[loop1].x++; // Move The Enemy Right
}

if ((enemy[loop1].x>player.x) && (enemy[loop1].fy==enemy[loop1].y*40))
{
enemy[loop1].x--; // Move The Enemy Left
}

if ((enemy[loop1].y<player.y) && (enemy[loop1].fx==enemy[loop1].x*60))
{
enemy[loop1].y++; // Move The Enemy Down
}

if ((enemy[loop1].y>player.y) && (enemy[loop1].fx==enemy[loop1].x*60))
{
enemy[loop1].y--; // Move The Enemy Up
}
```

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)) // If Our Delay Is Done And Player Doesn't Have
Hourglass
{
delay=0; // Reset The Delay Counter Back To Zero
```

```
for (loop2=0; loop2<(stage*level); loop2++) // Loop Through All The Enemies
{
```

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[loop2].x*60) // Is Fine Position On X Axis Lower Than Intended
Position?
{
enemy[loop2].fx+=steps[adjust]; // If So, Increase Fine Position On X Axis
enemy[loop2].spin+=steps[adjust]; // Spin Enemy Clockwise
}
if (enemy[loop2].fx>enemy[loop2].x*60) // Is Fine Position On X Axis Higher Than Intended
Position?
{
enemy[loop2].fx-=steps[adjust]; // If So, Decrease Fine Position On X Axis
enemy[loop2].spin-=steps[adjust]; // Spin Enemy Counter Clockwise
}
if (enemy[loop2].fy<enemy[loop2].y*40) // Is Fine Position On Y Axis Lower Than Intended
Position?
{
enemy[loop2].fy+=steps[adjust]; // If So, Increase Fine Position On Y Axis
enemy[loop2].spin+=steps[adjust]; // Spin Enemy Clockwise
}
if (enemy[loop2].fy>enemy[loop2].y*40) // Is Fine Position On Y Axis Higher Than Intended
Position?
{
enemy[loop2].fy-=steps[adjust]; // If So, Decrease Fine Position On Y Axis
enemy[loop2].spin-=steps[adjust]; // Spin Enemy Counter Clockwise
}
}
}
```

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 for PlaySound() and the timer to work, you have to include the WINMM.LIB file under PROJECT / SETTINGS / LINK in Visual C++. Winmm.lib is the Windows Multimedia Library. If you don't include this library, you will get error messages when you try to compile the program.

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

if (lives==0) // Are We Out Of Lives?
{
gameover=TRUE; // If So, gameover Becomes TRUE
}

ResetObjects(); // Reset Player / Enemy Positions
PlaySound("Data/Die.wav", NULL, SND_SYNC); // Play The Death Sound
}
}
```

Now we can move the player. In the first line of code below we check to see if the right arrow is being pressed, player.x is less than 10 (don't want to go off the grid), that player.fx equals player.x times 60 (lined up with a grid crossing on the x-axis, and that player.fy equals player.y times 40 (player is lined up with a grid crossing on the y-axis).

If we didn't make sure the player was at a crossing, and we allowed the player to move anyways, the player would cut right through the middle of boxes, just like the enemies would have done if we didn't make sure they were lined up with a vertical or horizontal line. Checking this also makes sure the player is done moving before we move to a new location.

If the player is at a grid crossing (where a vertical and horizontal lines meet) and he's not to far right, we mark the current horizontal line that we are on as being traced over. We then increase the player.x value by one, causing the new player position to be one box to the right.

We do the same thing while moving left, down and up. When moving left, we make sure the player wont be going off the left side of the grid. When moving down we make sure the player wont be leaving the bottom of the grid, and when moving up we make sure the player doesn't go off the top of the grid.

When moving left and right we make the horizontal line (hline[ ] [ ]) under us TRUE meaning it's been traced. When moving up and down we make the vertical line (vline[ ] [ ]) under us TRUE meaning it has been traced.

```
if (keys[VK_RIGHT] && (player.x<10) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
hline[player.x][player.y]=TRUE; // Mark The Current Horizontal Border As Filled
player.x++; // Move The Player Right
}
if (keys[VK_LEFT] && (player.x>0) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
player.x--; // Move The Player Left
hline[player.x][player.y]=TRUE; // Mark The Current Horizontal Border As Filled
}
if (keys[VK_DOWN] && (player.y<10) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
vline[player.x][player.y]=TRUE; // Mark The Current Verticle Border As Filled
player.y++; // Move The Player Down
}
if (keys[VK_UP] && (player.y>0) && (player.fx==player.x*60) && (player.fy==player.y*40))
{
player.y--; // Move The Player Up
vline[player.x][player.y]=TRUE; // Mark The Current Verticle Border As Filled
}
```

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 Fine Position On X Axis Lower Than Intended Position?
{
player.fx+=steps[adjust]; // If So, Increase The Fine X Position
}
if (player.fx>player.x*60) // Is Fine Position On X Axis Greater Than Intended Position?
{
player.fx-=steps[adjust]; // If So, Decrease The Fine X Position
}
if (player.fy<player.y*40) // Is Fine Position On Y Axis Lower Than Intended Position?
{
player.fy+=steps[adjust]; // If So, Increase The Fine Y Position
}
if (player.fy>player.y*40) // Is Fine Position On Y Axis Lower Than Intended Position?
{
player.fy-=steps[adjust]; // If So, Decrease The Fine Y Position
}
}
```

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 // Otherwise
{
if (keys[' ']) // If Spacebar Is Being Pressed
{
gameover=FALSE; // gameover Becomes FALSE
filled=TRUE; // filled Becomes TRUE
level=1; // Starting Level Is Set Back To One
level2=1; // Displayed Level Is Also Set To One
stage=0; // Game Stage Is Set To Zero
lives=5; // Lives Is Set To Five
}
}
```

The code below checks to see if the filled flag is TRUE (meaning the grid has been filled in). filled can be set to TRUE one of two ways. Either the grid is filled in completely and filled becomes TRUE or the game has ended but the spacebar was pressed to restart it (code above).

If filled is TRUE, the first thing we do is play the cool level complete tune. I've already explained how PlaySound() works. This time we'll be playing the Complete .WAV file in the DATA directory. Again, we use SND_SYNC so that there is a delay before the game starts on the next stage.

After the sound has played, we increase stage by one, and check to make sure stage isn't greater than 3. If stage is greater than 3 we set stage to 1, and increase the internal level and visible level by one.

If the internal level is greater than 3 we set the internal leve (level) to 3, and increase lives by 1. If you're amazing enough to get past level 3 you deserve a free life :). After increasing lives we check to make sure the player doesn't have more than 5 lives. If lives is greater than 5 we set lives back to 5.

```
if (filled) // Is The Grid Filled In?
{
PlaySound("Data/Complete.wav", NULL, SND_SYNC); // If So, Play The Level Complete Sound
stage++; // Increase The Stage
if (stage>3) // Is The Stage Higher Than 3?
{
stage=1; // If So, Set The Stage To One
level++; // Increase The Level
level2++; // Increase The Displayed Level
if (level>3) // Is The Level Greater Than 3?
{
level=3; // If So, Set The Level To 3
lives++; // Give The Player A Free Life
if (lives>5) // Does The Player Have More Than 5 Lives?
{
lives=5; // If So, Set Lives To Five
}
}
}
```

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(); // Reset Player / Enemy Positions

for (loop1=0; loop1<11; loop1++) // Loop Through The Grid X Coordinates
{
for (loop2=0; loop2<11; loop2++) // Loop Through The Grid Y Coordinates
{
if (loop1<10) // If X Coordinate Is Less Than 10
{
hline[loop1][loop2]=FALSE; // Set The Current Horizontal Value To FALSE
}
if (loop2<10) // If Y Coordinate Is Less Than 10
{
vline[loop1][loop2]=FALSE; // Set The Current Vertical Value To 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 On The Screen
if ((player.fx==hourglass.x*60) && (player.fy==hourglass.y*40) && (hourglass.fx==1))
{
// Play Freeze Enemy Sound
PlaySound("Data/freeze.wav", NULL, SND_ASYNC | SND_LOOP);
hourglass.fx=2; // Set The hourglass fx Variable To Two
hourglass.fy=0; // Set The hourglass fy Variable To Zero
}
```

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 The Player Clockwise
if (player.spin>360.0f) // Is The spin Value Greater Than 360?
{
player.spin-=360; // If So, Subtract 360
}
```

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 The Hourglass Counter Clockwise
if (hourglass.spin<0.0f) // Is The spin Value Less Than 0?
{
hourglass.spin+=360.0f; // If So, Add 360
}
```

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]; // Increase The hourglass fy Variable
if ((hourglass.fx==0) && (hourglass.fy>6000/level)) // Is The hourglass fx Variable Equal To 0
And The fy
{ // Variable Greater Than 6000 Divided By The Current Level?
PlaySound("Data/hourglass.wav", NULL, SND_ASYNC); // If So, Play The Hourglass Appears Sound
hourglass.x=rand()%10+1; // Give The Hourglass A Random X Value
hourglass.y=rand()%11; // Give The Hourglass A Random Y Value
hourglass.fx=1; // Set hourglass fx Variable To One (Hourglass Stage)
hourglass.fy=0; // Set hourglass fy Variable To Zero (Counter)
}
```

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 Variable Equal To 1
And The fy
{ // Variable Greater Than 6000 Divided By The Current Level?
hourglass.fx=0; // If So, Set fx To Zero (Hourglass Will Vanish)
hourglass.fy=0; // Set fy to Zero (Counter Is Reset)
}
```

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 The hourglass fx Variable Equal To
2 And The fy
{ // Variable Greater Than 500 Plus 500 Times The Current Level?
PlaySound(NULL, NULL, 0); // If So, Kill The Freeze Sound
hourglass.fx=0; // Set hourglass fx Variable To Zero
hourglass.fy=0; // Set hourglass fy Variable To Zero
}
```

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++; // Increase The Enemy Delay Counter
}
}

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

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**)

Lesson 22
Bump-Mapping, Multi-Texturing & Extensions

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> // 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
#include "glext.h" // Header File For Multitexturing
#include <string.h> // Header File For The String Library
#include <math.h> // Header File For The Math Library
```

The GLfloat MAX_EMBOSS specifies the "strength" of the Bump Mapping-Effect. Larger values strongly enhance the effect, but reduce visual quality to the same extent by leaving so-called "artefacts" at the edges of the surfaces.

```
#define MAX_EMBOSS (GLfloat)0.01f // Maximum Emboss-Translate. Increase To Get Higher Immersion
```

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 Disable ARB Extensions Entirely
// #define EXT_INFO // Uncomment To See Your Extensions At Start-Up?
#define MAX_EXTENSION_SPACE 10240 // Characters For Extension-Strings
#define MAX_EXTENSION_LENGTH 256 // Maximum Characters In One Extension-String
bool multitextureSupported=false; // Flag Indicating Whether Multitexturing Is Supported
bool useMultitexture=true; // Use It If It Is Supported?
GLint maxTexelUnits=1; // Number Of Texel-Pipelines. This Is At Least 1.
```

The following lines are needed to "link" the extensions to C++ function calls. Just treat the PFN-who-ever-reads-this as pre-defined datatype able to describe function calls. Since we are unsure if we'll get the functions to these prototypes, we set them to NULL. The commands glMultiTexCoordifARB map to the well-known glTexCoordif, specifying i-dimensional texture-coordinates. Note that these can totally substitute the glTexCoordif-commands. Since we only use the GLfloat-version, we only need prototypes for the commands ending with an "f". Other are also available ("fv", "i", etc.). The last two prototypes are to set the active texture-unit that is currently receiving texture-bindings (glActiveTextureARB()) and to determine which texture-unit is associated with the ArrayPointer-command (a.k.a. Client-Subset, thus glClientActiveTextureARB). By the way: ARB is an abbreviation for "Architectural Review Board". Extensions with ARB in their name are not required by an OpenGL-conformant implementation, but they are expected to be widely supported. Currently, only the 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 ommitted are GDI-context handles etc.

```
PFNGLMULTITEXCOORD1FARBPROC glMultiTexCoord1fARB = NULL;
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLMULTITEXCOORD3FARBPROC glMultiTexCoord3fARB = NULL;
PFNGLMULTITEXCOORD4FARBPROC glMultiTexCoord4fARB = NULL;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB= NULL;
```

We need global variables:

- filter specifies what filter to use. Refer to Lesson 06. We'll usually just take GL_LINEAR, so we initialise with 1.
- texture holds our base-texture, three times, one per filter.
- bump holds our bump maps
- invbump holds our inverted bump maps. This is explained later on in a theory-section.
- The Logo-things hold textures for several billboards that will be added to rendering output as a final pass.
- The Light...-stuff contains data on our OpenGL light-source.

```
GLuint filter=1; // Which Filter To Use
GLuint texture[3]; // Storage For 3 Textures
GLuint bump[3]; // Our Bumpmappings
GLuint invbump[3]; // Inverted Bumpmaps
GLuint glLogo; // Handle For OpenGL-Logo
GLuint multiLogo; // Handle For Multitexture-Enabled-Logo
GLfloat LightAmbient[] = { 0.2f, 0.2f, 0.2f}; // Ambient Light Is 20% White
GLfloat LightDiffuse[] = { 1.0f, 1.0f, 1.0f}; // Diffuse Light Is White
GLfloat LightPosition[] = { 0.0f, 0.0f, 2.0f}; // Position Is Somewhat In Front Of Screen
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 Tesselation Of The Cube Is Only Absolute Minimum.

GLfloat data[]= {
// FRONT FACE
0.0f, 0.0f, -1.0f, -1.0f, +1.0f,
1.0f, 0.0f, +1.0f, -1.0f, +1.0f,
1.0f, 1.0f, +1.0f, +1.0f, +1.0f,
0.0f, 1.0f, -1.0f, +1.0f, +1.0f,
// BACK FACE
1.0f, 0.0f, -1.0f, -1.0f, -1.0f,
1.0f, 1.0f, -1.0f, +1.0f, -1.0f,
0.0f, 1.0f, +1.0f, +1.0f, -1.0f,
0.0f, 0.0f, +1.0f, -1.0f, -1.0f,
// Top Face
0.0f, 1.0f, -1.0f, +1.0f, -1.0f,
0.0f, 0.0f, -1.0f, +1.0f, +1.0f,
1.0f, 0.0f, +1.0f, +1.0f, +1.0f,
1.0f, 1.0f, +1.0f, +1.0f, -1.0f,
// Bottom Face
1.0f, 1.0f, -1.0f, -1.0f, -1.0f,
0.0f, 1.0f, +1.0f, -1.0f, -1.0f,
0.0f, 0.0f, +1.0f, -1.0f, +1.0f,
1.0f, 0.0f, -1.0f, -1.0f, +1.0f,
// Right Face
1.0f, 0.0f, +1.0f, -1.0f, -1.0f,
1.0f, 1.0f, +1.0f, +1.0f, -1.0f,
0.0f, 1.0f, +1.0f, +1.0f, +1.0f,
0.0f, 0.0f, +1.0f, -1.0f, +1.0f,
// Left Face
0.0f, 0.0f, -1.0f, -1.0f, -1.0f,
1.0f, 0.0f, -1.0f, -1.0f, +1.0f,
1.0f, 1.0f, -1.0f, +1.0f, +1.0f,
0.0f, 1.0f, -1.0f, +1.0f, -1.0f
};

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

The next block of code is to determine extension-support during run-time.

First, we can assume that we have a long string containing all supported extensions as '\n'-seperated sub-strings. So all we need to do is to search for a '\n' and start comparing string with search until we encounter another '\n' or until string doesn't match search anymore. In the first case, return a true for "found", in the other case, take the next sub-string until you encounter the end of string. You'll have to watch a little bit at the beginning of string, since it does not begin with a newline-character.

By the way: A common rule is to ALWAYS check during runtime for availability of a given extension!

```
bool isInString(char *string, const char *search) {
int pos=0;
int maxpos=strlen(search)-1;
int len=strlen(string);
char *other;
for (int i=0; i<len; i++) {
if ((i==0) || ((i>1) && string[i-1]=='\n')) { // New Extension Begins Here!
other=&string[i];
pos=0; // Begin New Search
while (string[i]!='\n') { // Search Whole Extension-String
if (string[i]==search[pos]) pos++; // Next Position
if ((pos>maxpos) && string[i+1]=='\n') return true; // We Have A Winner!
i++;
}
}
}
return false; // Sorry, Not Found!
}
```

Now we have to fetch the extension-string and convert it to be '\n'-separated in order to search it for our desired extension. If we find a sub-string "GL_ARB_multitexture" in it, this feature is supported. But we only can use it, if __ARB_ENABLE is also true. Last but not least we need GL_EXT_texture_env_combine to be supported. This extension introduces new ways how the texture-units interact. We need this, since GL_ARB_multitexture only feeds the output from one texture unit to the one with the next higher number. So we rather check for this extension than using another complex blending equation (that would not exactly do the same effect!) If all extensions are supported and we are not overridden, we'll first determine how much texture-units are available, saving them in maxTexelUnits. Then we have to link the functions to our names. This is done by the wglGetProcAdress()-calls with a string naming the function call as parameter and a prototype-cast to ensure we'll get the correct function type.

```
bool initMultitexture(void) {
char *extensions;
extensions=strdup((char *) glGetString(GL_EXTENSIONS)); // Fetch Extension String
int len=strlen(extensions);
for (int i=0; i<len; i++) // Separate It By Newline Instead Of Blank
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 Multitexturing Supported?
&& __ARB_ENABLE // Override Flag
&& isInString(extensions,"GL_EXT_texture_env_combine")) // texture-environment-combining
supported?
{
glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB,&maxTexelUnits);
glMultiTexCoord1fARB = (PFNGLMULTITEXCOORD1FARBPROC) wglGetProcAddress("glMultiTexCoord1fARB");
glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC) wglGetProcAddress("glMultiTexCoord2fARB");
glMultiTexCoord3fARB = (PFNGLMULTITEXCOORD3FARBPROC) wglGetProcAddress("glMultiTexCoord3fARB");
glMultiTexCoord4fARB = (PFNGLMULTITEXCOORD4FARBPROC) wglGetProcAddress("glMultiTexCoord4fARB");
glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC) wglGetProcAddress("glActiveTextureARB");
glClientActiveTextureARB= (PFNGLCLIENTACTIVETEXTUREARBPROC)
wglGetProcAddress("glClientActiveTextureARB");

#ifdef EXT_INFO
MessageBox(hWnd,"The GL_ARB_multitexture extension will be used.","feature supported!",MB_OK |
MB_ICONINFORMATION);
#endif

return true;
}
useMultitexture=false; // We Can't Use It If It Isn't Supported!
return false;
}
```

InitLights() just initialises OpenGL-Lighting and is called by InitGL() later on.

```
void initLights(void) {
        glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient); // Load Light-Parameters into GL_LIGHT1
        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() { // Load Bitmaps And Convert To Textures
bool status=true; // Status Indicator
AUX_RGBImageRec *Image=NULL; // Create Storage Space For The Texture
char *alpha=NULL;

// Load The Tile-Bitmap for Base-Texture
if (Image=auxDIBImageLoad("Data/Base.bmp")) {
glGenTextures(3, texture); // Create Three Textures

// 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_RGB, GL_UNSIGNED_BYTE,
Image->data);

// 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_RGB, GL_UNSIGNED_BYTE,
Image->data);

// 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_NEAREST);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB8, Image->sizeX, Image->sizeY, GL_RGB, GL_UNSIGNED_BYTE,
Image->data);
}
else status=false;

if (Image) { // If Texture Exists
if (Image->data) delete Image->data; // If Texture Image Exists
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 RGB By 50%, So That We Have Only
glPixelTransferf(GL_GREEN_SCALE,0.5f); // Half Intenstity
glPixelTransferf(GL_BLUE_SCALE,0.5f);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP); // No Wrapping, Please!
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
glGenTextures(3, bump); // Create Three Textures

// 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 The Bumpmap
Image->data[i]=255-Image->data[i];

glGenTextures(3, invbump); // Create Three Textures

// Create Nearest Filtered Texture
>…<

// Create Linear Filtered Texture
>…<

// Create MipMapped Texture
>…<
}
else status=false;
if (Image) { // If Texture Exists
if (Image->data) delete Image->data; // If Texture Image Exists
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]; // Pick Only Red Value As Alpha!
if (!(Image=auxDIBImageLoad("Data/OpenGL.bmp"))) status=false;
for (a=0; a<Image->sizeX*Image->sizeY; a++) {
alpha[4*a]=Image->data[a*3]; // R
alpha[4*a+1]=Image->data[a*3+1]; // G
alpha[4*a+2]=Image->data[a*3+2]; // B
}

glGenTextures(1, &glLogo); // Create One Textures

// 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, GL_RGBA,
GL_UNSIGNED_BYTE, alpha);
delete alpha;
}
else status=false;

if (Image) { // If Texture Exists
if (Image->data) delete Image->data; // If Texture Image Exists
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 Memory For RGBA8-Texture
>…<
glGenTextures(1, &multiLogo); // Create One Textures
// Create Linear Filtered RGBA8-Texture
>…<
delete alpha;
}
```

```
else status=false;

if (Image) { // If Texture Exists
if (Image->data) delete Image->data; // If Texture Image Exists
delete Image;
Image=NULL;
}
return status; // Return The Status
}
```

Next comes nearly the only unmodified function ReSizeGLScene(). I've omitted it here. It is followed by a function doCube() that draws a cube, complete with normalized normals. Note that this version only feeds texture-unit #0, since glTexCoord2f(s,t) is the same thing as glMultiTexCoord2f(GL_TEXTURE0_ARB,s,t). Note also that the cube could be done using interleaved arrays, but this is definitely another issue. Note also that this cube CAN NOT be done using a display list, since display-lists seem to use an internal floating point accuracy different from GLfloat. Since this leads to several nasty effects, generally referred to as "decaling"-problems, I kicked display lists. I assume that a general rule for multipass algorithms is to do the entire geometry with or without display lists. So never dare mixing even if it seems to run on your hardware, since it won't run on any hardware!

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
// Resize And Initialize The GL Window
>…<

void doCube (void) {
int i;
glBegin(GL_QUADS);
// Front Face
glNormal3f( 0.0f, 0.0f, +1.0f);
for (i=0; i<4; i++) {
glTexCoord2f(data[5*i],data[5*i+1]);
glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// Back Face
glNormal3f( 0.0f, 0.0f,-1.0f);
for (i=4; i<8; i++) {
glTexCoord2f(data[5*i],data[5*i+1]);
glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// Top Face
glNormal3f( 0.0f, 1.0f, 0.0f);
for (i=8; i<12; i++) {
glTexCoord2f(data[5*i],data[5*i+1]);
glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// Bottom Face
glNormal3f( 0.0f,-1.0f, 0.0f);
for (i=12; i<16; i++) {
glTexCoord2f(data[5*i],data[5*i+1]);
glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// Right Face
glNormal3f( 1.0f, 0.0f, 0.0f);
for (i=16; i<20; i++) {
glTexCoord2f(data[5*i],data[5*i+1]);
glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// Left Face
glNormal3f(-1.0f, 0.0f, 0.0f);
for (i=20; i<24; i++) {
glTexCoord2f(data[5*i],data[5*i+1]);
glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
glEnd();
}
```

Time to initialize OpenGL. All as in Lesson 06, except that I call initLights() instead of setting them here. Oh, and of course I'm calling Multitexture-setup, here!

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
multitextureSupported=initMultitexture();
if (!LoadGLTextures()) return false; // Jump To Texture Loading Routine
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
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 Calculations

initLights(); // Initialize OpenGL Light
return true // Initialization Went OK
}
```

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) x Dl x Dm

- L is light vector
- N is normal vector
- Dl is light diffuse color
- Dm is material diffuse color
- Bump Mapping changes N per pixel
- Emboss Bump Mapping approximates (L*N)

Approximate Diffuse Factor L*N

Texture Map Represents Heightfield

- [0,1] represents range of bump function
- First derivate represents slope m (Note that m is only 1D. Imagine m to be the inf.-norm of grad(s,t) to a given set of coordinates (s,t)!, rem.: Jens)
- m increases / decreases base diffuse factor Fd
- (Fd+m) approximates (L*N) per pixel

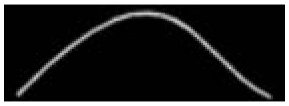
Approximate Derivative

Embossing Approximates Derivative

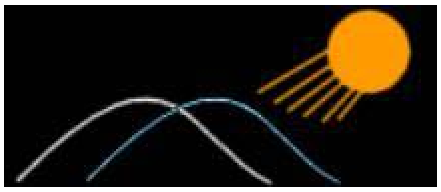
- Lookup height H0 at point (s,t)
- Lookup height H1 at point slightly perturbed toward light source (s+ds,t+dt)

- Subtract original height H0 from perturbed height H1
- Difference represents instantaneous slope m=H1-H0

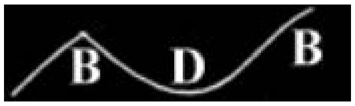
Compute The Bump



1) Original bump (H0).



2) Original bump (H0) overlaid with second bump (H1) slightly perturbed toward light source.



3) Substract original bump from second (H0-H1). This leads to brightened (B) and darkened (D) areas.

Compute The Lighting

Evaluate Fragment Color Cf

- Cf = (L*N) x Dl x Dm
- (L*N) ~ (Fd + (H1-H0))
- Dm x Dl is encoded in surface texture Ct. Could control Dl seperately, if you're clever. (we control it using OpenGL-Lighting!, rem.: Jens)
- Cf = (Fd + (H0-H1) x Ct

Is That All? It's So Easy!

We're Not Quite Done Yet. We Still Must:

- Build a texture (using a painting program, rem.: Jens)
- Calculate texture coordinate offsets (ds,dt)
- Calculate diffuse Factor Fd (is controlled using OpenGL-Lighting!, rem.: Jens)
- Both are derived from normal N and light vector L (in our case, only (ds,dt) are calculated explicitly!, rem.: Jens)
- Now we have to do some math

Building A Texture

Conserve Textures!

- Current multitexture-hardware only supports two textures! (By now, not true anymore, but nevertheless you should read this!, rem.: Jens)
- Bump Map in ALPHA channel (not the way we do it, could implement it yourself as an exercise if you have TNT-chipset rem.: Jens)
- Maximum bump = 1.0
- Level ground = 0.5
- Maximum depression = 0.0
- Surface color in RGB channels
- Set internal format to GL_RGBA8 !!

Calculate Texture Offsets

Rotate Light Vector Into Normal Space

- Need Normal coordinate system
- Derive coordinate system from normal and “up” vector (we pass the texCoord directions to our offset generator explicitly, rem.: Jens)
- Normal is z-axis
- Cross-product is x-axis
- Throw away "up" vector, derive y-axis as cross-product of x- and z-axis
- Build 3x3 matrix Mn from axes
- Transform light vector into normal space.(Mn is also called an orthonormal basis. Think of Mn*v as to "express" v in means of a basis describing tangent space rather than in means of the standard basis. Note also that orthonormal bases are invariant against-scaling resulting in no loss of normalization when multiplying vectors! rem.: Jens)

Calculate Texture Offsets (Cont’d)

Use Normal-Space Light Vector For Offsets

- L’ = Mn x 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]; // Homogenous Coordinate
}
```

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:

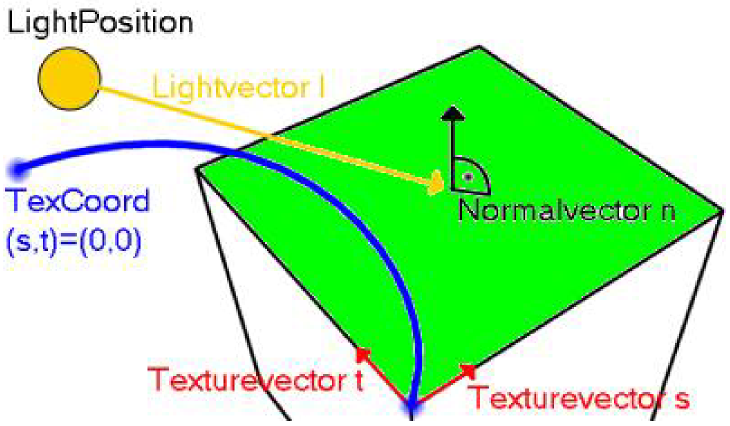
- We use OBJECT COORDINATES, this means we don't apply the modelview matrix to our calculations. This has a nasty side-effect: since we want to rotate the cube, object-coordinates of the cube don't change, world-coordinates (also referred to as eye-coordinates) do. But our light-position should not be 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_EMBOSS: ds = s*v*MAX_EMBOSS, dt=t*v*MAX_EMBOSS. Note that s, t and v are vectors while MAX_EMBOSS isn't.
- Add the offset to the texture-coordinates in pass 2.

Why this is good:

- Fast (only needs one squareroot and a couple of MULs per vertex)!
- Looks very nice!
- This works with all surfaces, not just planes.
- This runs on all accelerators.
- Is glBegin/glEnd friendly: Does not need any "forbidden" GL-commands.

Drawback:

- Not fully physical correct.
- Leaves minor artefacts.



This figure shows where our vectors are located. You can get t and s by simply subtracting adjacent vertices, but be sure to have them point in the right direction and to normalize them. The blue spot marks the vertex where texCoord2f(0.0f,0.0f) is mapped to.

End Theory (Emboss Bump Mapping Algorithms)

Let's have a look to texture-coordinate offset generation, first. The function is called SetUpBumps(), since this actually is what it does:

```
// Sets Up The Texture-Offsets
// n : Normal On Surface. Must Be Of Length 1
// c : Current Vertex On Surface
// l : Lightposition
// s : Direction Of s-Texture-Coordinate In Object Space (Must Be Normalized!)
// t : Direction Of t-Texture-Coordinate In Object Space (Must Be Normalized!)
void SetUpBumps(GLfloat *n, GLfloat *c, GLfloat *l, GLfloat *s, GLfloat *t) {
GLfloat v[3]; // Vertex From Current Position To Light
GLfloat lenQ; // Used To Normalize
// 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_EMBOSS;
c[1]=(t[0]*v[0]+t[1]*v[1]+t[2]*v[2])*MAX_EMBOSS;
```

Doesn't look that complicated anymore, eh? But theory is necessary to understand and control this effect. (I learned THAT myself during writing this tutorial).

I always like logos to be displayed while presentational programs are running. We'll have two of them right now. Since a call to doLogo() resets the GL_MODELVIEW-matrix, this has to be called as final rendering pass.

This function displays two logos: An OpenGL-Logo and a multitexture-Logo, if this feature is enabled. The logos are alpha-blended and are sort of semi-transparent. Since they have an alpha-channel, I blend them using GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, as suggested by all OpenGL-documentation. Since they are all co-planar, we do not have to z-sort them before. The numbers that are used for the vertices are "empirical" (a.k.a. try-and-error) to place them neatly into the screen edges. We'll have to enable blending and disable lighting to avoid nasty effects. To ensure they're in front of all, just reset the GL_MODELVIEW-matrix and set depth-function to GL_ALWAYS.

```
void doLogo(void) {
// MUST CALL THIS LAST!!!, Billboards The Two Logos
glDepthFunc(GL_ALWAYS);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
glDisable(GL_LIGHTING);
glLoadIdentity();
glBindTexture(GL_TEXTURE_2D,glLogo);
glBegin(GL_QUADS);
glTexCoord2f(0.0f,0.0f); glVertex3f(0.23f, -0.4f,-1.0f);
glTexCoord2f(1.0f,0.0f); glVertex3f(0.53f, -0.4f,-1.0f);
glTexCoord2f(1.0f,1.0f); glVertex3f(0.53f, -0.25f,-1.0f);
glTexCoord2f(0.0f,1.0f); glVertex3f(0.23f, -0.25f,-1.0f);
glEnd();
if (useMultitexture) {
glBindTexture(GL_TEXTURE_2D,multiLogo);
glBegin(GL_QUADS);
glTexCoord2f(0.0f,0.0f); glVertex3f(-0.53f, -0.25f,-1.0f);
glTexCoord2f(1.0f,0.0f); glVertex3f(-0.33f, -0.25f,-1.0f);
glTexCoord2f(1.0f,1.0f); glVertex3f(-0.33f, -0.15f,-1.0f);
glTexCoord2f(0.0f,1.0f); glVertex3f(-0.53f, -0.15f,-1.0f);
glEnd();
}
}
```

Here comes the function for doing the bump mapping without multitexturing. It's a three-pass implementation. As a first step, the GL_MODELVIEW matrix is inverted by applying to the identity-matrix all steps later applied to the GL_MODELVIEW in reverse order and inverted. The result is a matrix that "undoes" the GL_MODELVIEW if applied to an object. We fetch it from OpenGL by simply using glGetFloatv(). Remember that the matrix has to be an array of 16 and that the matrix is "transposed"!

By the way: If you don't exactly know how the modelview was built, consider using world-space, since matrix-inversion is complicated and costly. But if you're doing large amounts of vertices inverting the modelview with a more generalized approach could be faster.

```
bool doMesh1TexelUnits(void) {
GLfloat c[4]={0.0f,0.0f,0.0f,1.0f}; // Holds Current Vertex
GLfloat n[4]={0.0f,0.0f,0.0f,1.0f}; // Normalized Normal Of Current Surface
GLfloat s[4]={0.0f,0.0f,0.0f,1.0f}; // s-Texture Coordinate Direction, Normalized
GLfloat t[4]={0.0f,0.0f,0.0f,1.0f}; // t-Texture Coordinate Direction, Normalized
GLfloat l[4]; // Holds Our Lightposition To Be Transformed Into Object Space
GLfloat Minv[16]; // Holds The Inverted Modelview Matrix To Do So
int i;

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer

// Build Inverse Modelview Matrix First. This Substitutes One Push/Pop With One
glLoadIdentity();
// 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; // Homogenous Coordinate
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();
```

<u>Third Pass:</u>

- 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();
}
```

<u>Last Pass:</u>

- 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; // Keep Going
}
```

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 Current Vertex
GLfloat n[4]={0.0f,0.0f,0.0f,1.0f}; // Normalized Normal Of Current Surface
GLfloat s[4]={0.0f,0.0f,0.0f,1.0f}; // s-Texture Coordinate Direction, Normalized
GLfloat t[4]={0.0f,0.0f,0.0f,1.0f}; // t-Texture Coordinate Direction, Normalized
GLfloat l[4]; // Holds Our Lightposition To Be Transformed Into Object Space
GLfloat Minv[16]; // Holds The Inverted Modelview Matrix To Do So
int i;

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer

// Build Inverse Modelview Matrix First. This Substitutes One Push/Pop With One
glLoadIdentity();
// 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; // Homogenous Coordinate
VMatMult(Minv,l);
```

First Pass:

- No Blending
- No Lighting

Set up the texture-combiner 0 to

- Use bump-texture
- Use not-offset texture-coordinates
- Texture-Operation GL_REPLACE, resulting in texture just being drawn

Set up the texture-combiner 1 to

- Offset texture-coordinates
- Texture-Operation GL_ADD, which is the multitexture-equivalent to ONE, ONE- blending.

This will render a cube consisting out of the grey-scale erode map.

```
// TEXTURE-UNIT #0
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, bump[filter]);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);

// TEXTURE-UNIT #1
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
```

```
glBindTexture(GL_TEXTURE_2D, invbump[filter]);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_ADD);

// General Switches
glDisable(GL_BLEND);
glDisable(GL_LIGHTING);
```

Now just render the faces one by one, as already seen in doMesh1TexelUnits(). Only new thing: Uses glMultiTexCoor2fARB() instead of just glTexCoord2f(). Note that you must specify which texture-unit you mean by the first parameter, which must be GL_TEXTUREi_ARB with i in [0..31]. (What hardware has 32 texture-units? And what for?)

```
glBegin(GL_QUADS);
// Front Face
n[0]=0.0f;
n[1]=0.0f;
n[2]=1.0f;
s[0]=1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=0; i<4; i++) {
c[0]=data[5*i+2];
c[1]=data[5*i+3];
c[2]=data[5*i+4];
SetUpBumps(n,c,l,s,t);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+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);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,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);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,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);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,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);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,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);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,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();
```

Second Pass

- Use the base-texture
- Enable Lighting
- No offset texturre-coordinates => reset GL_TEXTURE-matrix
- Reset texture environment to GL_MODULATE in order to do OpenGLLighting (doesn't work otherwise!)

This will render our complete bump-mapped cube.

```
glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);
if (!emboss) {
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glBindTexture(GL_TEXTURE_2D,texture[filter]);
glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
glEnable(GL_BLEND);
glEnable(GL_LIGHTING);
doCube();
}
```

Last Pass

- Update Geometry (esp. rotations)
- Do The Logos

```
xrot+=xspeed;
yrot+=yspeed;
if (xrot>360.0f) xrot-=360.0f;
if (xrot<0.0f) xrot+=360.0f;
if (yrot>360.0f) yrot-=360.0f;
if (yrot<0.0f) yrot+=360.0f;
```

```
/* LAST PASS: Do The Logos! */
doLogo();
return true; // Keep Going
}
```

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 The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
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; // Keep Going
}
```

All the drawGLScene() function has to do is to determine which doMesh-function to call:

```
bool DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
if (bumps) {
if (useMultitexture && maxTexelUnits>1)
return doMesh2TexelUnits();
else return doMesh1TexelUnits();
}
else return doMeshNoBumps();
}
```

Kills the GLWindow, not modified (thus omitted):

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
>…<
```

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 For This Window
UINT uMsg, // Message For This Window
WPARAM wParam, // Additional Message Information
LPARAM lParam) // Additional Message Information
>…<
```

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, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{

>…<

if (keys['E'])
{
keys['E']=false;
emboss=!emboss;
}

if (keys['M'])
{
keys['M']=false;
useMultitexture=((!useMultitexture) && multitextureSupported);
}

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

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**)

Lesson 23
Sphere Mapping Quadrics In OpenGL

Sphere Environment Mapping is a quick way to add a reflection to a metallic or reflective object in your scene. Although it is not as accurate as real life or as a Cube Environment Map, it is a whole lot faster! We'll be using the code from lesson eighteen (Quadrics) for the base of this tutorial. Also we're not using any of the same texture maps, we're going to use one sphere map, and one background image.

Before we start... The "red book" defines a Sphere map as a picture of the scene on a metal ball from infinite distance away and infinite focal point. Well that is impossible to do in real life. The best way I have found to create a good sphere map image without using a Fish eye lens is to use Adobe's Photoshop program.

Creating a Sphere Map In Photoshop:

First you will need a picture of the environment you want to map onto the sphere. Open the picture in Adobe Photoshop and select the entire image. Copy the image and create a new PSD (Photoshop Format) the new image should be the same size as the image we just copied. Paste a copy of the image into the new window we've created. The reason we make a copy is so Photoshop can apply its filters. Instead of copying the image you can select mode from the drop down menu and choose RGB mode. All of the filters should then be available.

Next we need to resize the image so that the image dimensions are a power of 2. Remember that in order to use an image as a texture the image needs to be 128x128, 256x256, etc. Under the image menu, select image size, uncheck the constraint proportions checkbox, and resize the image to a valid texture size. If your image is 100X90, it's better to make the image 128x128 than 64x64. Making the image smaller will lose alot of detail.

The last thing we do is select the filter menu, select distort and apply a spherize modifier. You should see that the center of the picture is blown up like a balloon, now in normal sphere maps the outer area will be blackened out, but it doesn't really matter. Save a copy of the image as a .BMP and you're ready to code!

We don't add any new global variables this time but we do modify the texture array to hold 6 textures.

```
GLuint texture[6]; // Storage For 6 Textures
```

The next thing I did was modify the LoadGLTextures() function so we can load in 2 bitmaps and create 3 filters. (Like we did in the original texturing tutorials). Basically we loop through twice and create 3 textures each time using a different filtering mode. Almost all of this code is new or modified.

```
int LoadGLTextures() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator

AUX_RGBImageRec *TextureImage[2]; // Create Storage Space For The Texture

memset(TextureImage,0,sizeof(void *)*2); // Set The Pointer To NULL

// Load The Bitmap, Check For Errors, If Bitmap's Not Found Quit
if ((TextureImage[0]=LoadBMP("Data/BG.bmp")) && // Background Texture
(TextureImage[1]=LoadBMP("Data/Reflect.bmp"))) // Reflection Texture (Spheremap)
{
Status=TRUE; // Set The Status To TRUE

glGenTextures(6, &texture[0]); // Create Three Textures
for (int loop=0; loop<=1; loop++)
{
// Create Nearest Filtered Texture
glBindTexture(GL_TEXTURE_2D, texture[loop]); // Gen Tex 0 And 1
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);

// Create Linear Filtered Texture
glBindTexture(GL_TEXTURE_2D, texture[loop+2]); // Gen Tex 2, 3 And 4
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, TextureImage[loop]->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);

// Create MipMapped Texture
glBindTexture(GL_TEXTURE_2D, texture[loop+4]); // Gen Tex 4 and 5
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAREST);
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY,
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
}
for (loop=0; loop<=1; loop++)
{
if (TextureImage[loop]) // If Texture Exists
{
if (TextureImage[loop]->data) // If Texture Image Exists
{
free(TextureImage[loop]->data); // Free The Texture Image Memory
}
free(TextureImage[loop]); // Free The Image Structure
}
}
}

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

We'll modify the cube drawing code a little. Instead of using 1.0 and -1.0 for the normal values, we'll use 0.5 and -0.5. By changing the value of the normal, you can zoom the reflection map in and out. If the normal value is high, the image being reflected will be bigger, and may appear blocky. By reducing the normal value to 0.5 and -0.5 the reflected image is zoomed out a bit so that the image reflecting off the cube isn't all blocky looking. Setting the normal value too low will create undesirable results.

```
GLvoid glDrawCube()
{
glBegin(GL_QUADS);
// Front Face
glNormal3f( 0.0f, 0.0f, 0.5f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
// Back Face
glNormal3f( 0.0f, 0.0f,-0.5f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Top Face
glNormal3f( 0.0f, 0.5f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
// Bottom Face
glNormal3f( 0.0f,-0.5f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
// Right Face
glNormal3f( 0.5f, 0.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Left Face
glNormal3f(-0.5f, 0.0f, 0.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
}
```

Now in InitGL we add two new function calls, these two calls set the texture generation mode for S and T to Sphere Mapping. The texture coordinates S, T, R & Q relate in a way to object coordinates x, y, z and w. If you are using a one-dimensional texture (1D) you will use the S coordinate. If your texture is two dimensional, you will use the S & T coordinates.

So what the following code does is tells OpenGL how to automatically generate the S and T coordinates for us based on the sphere-mapping formula. The R and Q coordinates are usually ignored. The Q coordinate can be used for advanced texture mapping extensions, and the R coordinate may become useful once 3D texture mapping has been added to OpenGL, but for now we will ignore the R & Q Coords. The S coordinate runs horizontally across the face of our polygon, the T coordinate runs vertically across the face of our polygon.

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set The Texture Generation Mode For S To
Sphere Mapping ( NEW )
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set The Texture Generation Mode For T To
Sphere Mapping ( NEW )
```

We're almost done! All we have to do is set up the rendering, I took out a few of the quadratic objects because they didn't work well with environment mapping. The first thing we need to do is enable texture generation. Then we select the reflective texture (sphere map) and draw our object. After all of the objects you want sphere-mapped have been drawn, you will want to disable texture generation, otherwise everything will be sphere mapped. We disable sphere-mapping before we draw the background scene (we don't want the background sphere mapped). You will notice that the bind texture commands may look fairly complex. All we're doing is selecting the filter to use when drawing our sphere map or the background image.

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View

glTranslatef(0.0f,0.0f,z);

glEnable(GL_TEXTURE_GEN_S); // Enable Texture Coord Generation For S ( NEW )
glEnable(GL_TEXTURE_GEN_T); // Enable Texture Coord Generation For T ( NEW )

glBindTexture(GL_TEXTURE_2D, texture[filter+(filter+1)]); // This Will Select A Sphere Map
glPushMatrix();
glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);
switch(object)
{
case 0:
glDrawCube();
break;
case 1:
glTranslatef(0.0f,0.0f,-1.5f); // Center The Cylinder
gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // A Cylinder With A Radius Of 0.5 And A Height Of
2
break;
case 2:
gluSphere(quadratic,1.3f,32,32); // Sphere With A Radius Of 1 And 16 Longitude/Latitude Segments
break;
case 3:
glTranslatef(0.0f,0.0f,-1.5f); // Center The Cone
gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32); // Cone With A Bottom Radius Of .5 And Height Of 2
break;
};

glPopMatrix();
glDisable(GL_TEXTURE_GEN_S); // Disable Texture Coord Generation ( NEW )
glDisable(GL_TEXTURE_GEN_T); // Disable Texture Coord Generation ( NEW )

glBindTexture(GL_TEXTURE_2D, texture[filter*2]); // This Will Select The BG Texture ( NEW )
glPushMatrix();
glTranslatef(0.0f, 0.0f, -24.0f);
glBegin(GL_QUADS);
glNormal3f( 0.0f, 0.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-13.3f, -10.0f, 10.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 13.3f, -10.0f, 10.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 13.3f, 10.0f, 10.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-13.3f, 10.0f, 10.0f);
glEnd();

glPopMatrix();

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

The last thing we have to do is update the spacebar section of code to reflect (No Pun Intended) the changes we made to the Quadratic objects being rendered. (We removed the discs)

```
if (keys[' '] && !sp)
```

```
{
sp=TRUE;
object++;
if(object>3)
object=0;
}
```

We're done! Now you can do some really impressive things with Environment mapping like making an almost accurate reflection of a room! I was planning on showing how to do Cube Environment Mapping in this tutorial too but my current video card does not support cube mapping. Maybe in a month or so after I buy a GeForce 2 :) Also I taught myself environment mapping (mostly because I couldnt find too much information on it) so if anything in this tutorial is inaccurate, Email Me or let NeHe know.

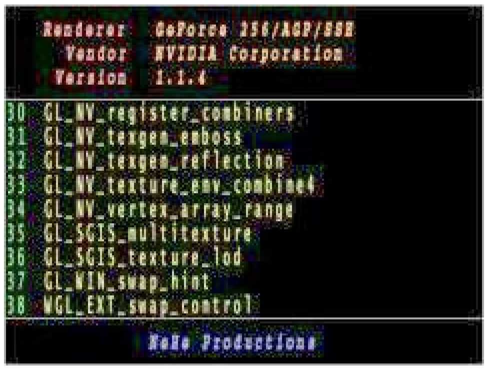
Thanks, and Good Luck!

Visit my site: http://www.tiptup.com/

GB Schmick (**TipTup**)

Jeff Molofee (**NeHe**)

Lesson 24
Tokens, Extensions, Scissor Testing And TGA Loading



```
Renderer   GeForce 256/AGP/SSE
  Vendor   NVIDIA Corporation
 Version   1.1.4
30  GL_NV_register_combiners
31  GL_NV_texgen_emboss
32  GL_NV_texgen_reflection
33  GL_NV_texture_env_combine4
34  GL_NV_vertex_array_range
35  GL_SGIS_multitexture
36  GL_SGIS_texture_lod
37  GL_WIN_swap_hint
38  WGL_EXT_swap_control
            NeHe Productions
```

This tutorial is far from visually stunning, but you will definitely learn a few new things by reading through it. I have had quite a few people ask me about extensions, and how to find out what extensions are supported on a particular brand of video card. This tutorial will teach you how to find out what OpenGL extensions are supported on any type of 3D video card.

I will also teach you how to scroll a portion of the screen without affecting any of the graphics around it using scissor testing. You will also learn how to draw line strips, and most importantly, in this tutorial we will drop the AUX library completely, along with Bitmap images. I will show you how to use Targa (TGA) images as textures. Not only are Targa files easy to work with and create, they support the ALPHA channel, which will allow you to create some pretty cool effects in future projects!

The first thing you should notice in the code below is that we no longer include the glaux header file (glaux.h). It is also important to note that the glaux.lib file can also be left out! We're not working with bitmaps anymore, so there's no need to include either of these files in our project.

Also, using glaux, I always received one warning message. Without glaux there should be zero errors, zero warnings.

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

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

bool keys[256]; // Array Used For The Keyboard Routine
bool active=TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
```

The first thing we need to do is add some variables. The first variable scroll will be used to scroll a portion of the screen up and down. The second variable maxtokens will be used to keep track of how many tokens (extensions) are supported by the video card.

base is used to hold the font display list.

swidth and sheight are used to grab the current window size. We use these two variable to help us calculate the scissor coordinates later in the code.

```
int scroll; // Used For Scrolling The Screen
int maxtokens; // Keeps Track Of The Number Of Extensions Supported
int swidth; // Scissor Width
int sheight; // Scissor Height

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

Now we create a structure to hold the TGA information once we load it in. The first variable imageData will hold a pointer to the data that makes up the image. bpp will hold the bits per pixel used in the TGA file (this value should be 24 or 32 bits depending on whether or not there is an alpha channel). The third variable width will hold the width of the TGA image. height will hold the height of the image, and texID will be used to keep track of the textures once they are built. The structure will be called TextureImage.

The line just after the structure (TextureImage textures[1]) sets aside storage for the one texture that we will be using in this program.

```
typedef struct // Create A Structure
{
GLubyte *imageData; // Image Data (Up To 32 Bits)
GLuint bpp; // Image Color Depth In Bits Per Pixel
GLuint width; // Image Width
GLuint height; // Image Height
GLuint texID; // Texture ID Used To Select A Texture
} TextureImage; // Structure Name

TextureImage textures[1]; // Storage For One Texture

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

Now for the fun stuff! This section of code will load in a TGA file and convert it into a texture for use in the program. One thing to note is that this code will only load 24 or 32 bit uncompressed TGA files. I had a hard enough time making the code work with both 24 and 32 bit TGA's :) I never said I was a genious. I'd like to point out that I did not write all of this code on my own. Alot of the really good ideas I got from reading through random sites on the net. I just took all the good ideas and combined them into code that works well with OpenGL. Not easy, not extremely difficult!

We pass two parameters to this section of code. The first parameter points to memory that we can store the texture in (*texture). The second parameter is the name of the file that we want to load (*filename).

The first variable TGAheader[ ] holds 12 bytes. We'll compare these bytes with the first 12 bytes we read from the TGA file to make sure that the file is indeed a Targa file, and not some other type of image.

TGAcompare will be used to hold the first 12 bytes we read in from the TGA file. The bytes in TGAcompare will then be compared with the bytes in TGAheader to make sure everything matches.

header[ ] will hold the first 6 IMPORTANT bytes from the header file (width, height, and bits per pixel).

The variable bytesPerPixel will store the result after we divide bits per pixel by 8, leaving us with the number of bytes used per pixel.

imageSize will store the number of bytes required to make up the image (width * height * bytes per pixel).

temp is a temporary variable that we will use to swap bytes later in the program.

The last variable type is a variable that I use to select the proper texture building params depending on whether or not the TGA is 24 or 32 bit. If the texture is 24 bit we need to use GL_RGB mode when we build the texture. If the TGA is 32 bit we need to add the Alpha component, meaning we have to use GL_RGBA (By default I assume the image is 32 bit by default that is why type is GL_RGBA).

```
bool LoadTGA(TextureImage *texture, char *filename) // Loads A TGA File Into Memory
{
GLubyte TGAheader[12]={0,0,2,0,0,0,0,0,0,0,0,0}; // Uncompressed TGA Header
GLubyte TGAcompare[12]; // Used To Compare TGA Header
GLubyte header[6]; // First 6 Useful Bytes From The Header
GLuint bytesPerPixel; // Holds Number Of Bytes Per Pixel Used In The TGA File
GLuint imageSize; // Used To Store The Image Size When Setting Aside Ram
GLuint temp; // Temporary Variable
GLuint type=GL_RGBA; // Set The Default GL Mode To RBGA (32 BPP)
```

The first line below opens the TGA file for reading. file is the handle we will use to point to the data within the file. the command fopen(filename, "rb") will open the file filename, and "rb" tells our program to open it for [r]eading in [b]inary mode!

The if statement has a few jobs. First off it checks to see if the file contains any data. If there is no data, NULL will be returned, the file will be closed with fclose(file), and we return false.

If the file contains information, we attempt to read the first 12 bytes of the file into TGAcompare. We break the line down like this: fread will read sizeof(TGAcompare) (12 bytes) from file into TGAcompare. Then we check to see if the number of bytes read is equal to sizeof(TGAcompare) which should be 12 bytes. If we were unable to read the 12 bytes into TGAcompare the file will close and false will be returned.

If everything has gone good so far, we then compare the 12 bytes we read into TGAcompare with the 12 bytes we have stored in TGAheader. If the bytes do not match, the file will close, and false will be returned.

Lastly, if everything has gone great, we attempt to read 6 more bytes into header (the important bytes). If 6 bytes are not available, again, the file will close and the program will return false.

```
FILE *file = fopen(filename, "rb"); // Open The TGA File

if( file==NULL || // Does File Even Exist?
fread(TGAcompare,1,sizeof(TGAcompare),file)!=sizeof(TGAcompare) || // Are There 12 Bytes To
Read?
memcmp(TGAheader,TGAcompare,sizeof(TGAheader))!=0 || // Does The Header Match What We Want?
fread(header,1,sizeof(header),file)!=sizeof(header)) // If So Read Next 6 Header Bytes
{
if (file == NULL) // Did The File Even Exist? *Added Jim Strong*
return false; // Return False
else
{
fclose(file); // If Anything Failed, Close The File
return false; // Return False
}
}
```

If everything went ok, we now have enough information to define some important variables. The first variable we want to define is width. We want width to equal the width of the TGA file. We can find out the TGA width by multiplying the value stored in header[1] by 256. We then add the lowbyte which is stored in header[0].

The height is calculated the same way but instead of using the values stored in header[0] and header[1] we use the values stored in header[2] and header[3].

After we have calculated the width and height we check to see if either the width or height is less than or equal to 0. If either of the two variables is less than or equal to zero, the file will be closed, and false will be returned.

We also check to see if the TGA is a 24 or 32 bit image. We do this by checking the value stored at header[4]. If the value is not 24 or 32 (bit), the file will be closed, and false will be returned.

In case you have not realized. A return of false will cause the program to fail with the message "Initialization Failed". Make sure your TGA is an uncompressed 24 or 32 bit image!

```
texture->width = header[1] * 256 + header[0]; // Determine The TGA Width (highbyte*256+lowbyte)
texture->height = header[3] * 256 + header[2]; // Determine The TGA Height
(highbyte*256+lowbyte)

if( texture->width <=0 || // Is The Width Less Than Or Equal To Zero
texture->height <=0 || // Is The Height Less Than Or Equal To Zero
(header[4]!=24 && header[4]!=32)) // Is The TGA 24 or 32 Bit?
{
fclose(file); // If Anything Failed, Close The File
return false; // Return False
}
```

Now that we have calculated the image width and height we need to calculate the bits per pixel, bytes per pixel and image size.

The value in header[4] is the bits per pixel. So we set bpp to equal header[4].

If you know anything about bits and bytes, you know that 8 bits makes a byte. To figure out how many bytes per pixel the TGA uses, all we have to do is divide bits per pixel by 8. If the image is 32 bit, bytesPerPixel will equal 4. If the image is 24 bit, bytesPerPixel will equal 3.

To calculate the image size, we multiply width * height * bytesPerPixel. The result is stored in imageSize. If the image was 100x100x32 bit our image size would be 100 * 100 * 32/8 which equals 10000 * 4 or 40000 bytes!

```
texture->bpp = header[4]; // Grab The TGA's Bits Per Pixel (24 or 32)
bytesPerPixel = texture->bpp/8; // Divide By 8 To Get The Bytes Per Pixel
imageSize = texture->width*texture->height*bytesPerPixel; // Calculate The Memory Required For
The TGA Data
```

Now that we know how many bytes our image is going to take, we need to allocate some memory. The first line below does the trick. imageData will point to a section of ram big enough to hold our image. malloc(imagesize) allocates the memory (sets memory aside for us to use) based on the amount of ram we request (imageSize).

The "if" statement has a few tasks. First it checks to see if the memory was allocated properly. If not, imageData will equal NULL, the file will be closed, and false will be returned.

If the memory was allocated, we attempt to read the image data from the file into the allocated memory. The line fread(texture->imageData, 1, imageSize, file) does the trick. fread means file read. imageData points to the memory we want to store the data in. 1 is the size of data we want to read in bytes (we want to read 1 byte at a time). imageSize is the total number of bytes we want to read. Because imageSize is equal to the total amount of ram required to hold the image, we end up reading in the entire image. file is the handle for our open file.

After reading in the data, we check to see if the amount of data we read in is the same as the value stored in imageSize. If the amount of data read and the value of imageSize is not the same, something went wrong. If any data was loaded, we will free it. (release the memory we allocated). The file will be closed, and false will be returned.

```
texture->imageData=(GLubyte *)malloc(imageSize); // Reserve Memory To Hold The TGA Data
```

```
if( texture->imageData==NULL || // Does The Storage Memory Exist?
fread(texture->imageData, 1, imageSize, file)!=imageSize) // Does The Image Size Match The
Memory Reserved?
{
if(texture->imageData!=NULL) // Was Image Data Loaded
free(texture->imageData); // If So, Release The Image Data

fclose(file); // Close The File
return false; // Return False
}
```

If the data was loaded properly, things are going good :) All we have to do now is swap the Red and Blue bytes. In OpenGL we use RGB (red, green, blue). The data in a TGA file is stored BGR (blue, green, red). If we didn't swap the red and blue bytes, anything in the picture that should be red would be blue and anything that should be blue would be red.

The first thing we do is create a loop (i) that goes from 0 to imageSize. By doing this, we can loop through all of the image data. Our loop will increase by steps of 3 (0, 3, 6, 9, etc) if the TGA file is 24 bit, and 4 (0, 4, 8, 12, etc) if the image is 32 bit. The reason we increase by steps is so that the value at i is always going to be the first byte ([b]lue byte) in our group of 3 or 4 bytes.

Inside the loop, we store the [b]lue byte in our temp variable. We then grab the red byte which is stored at texture->imageData[i+2] (Remember that TGAs store the colors as BGR[A]. B is i+0, G is i+1 and R is i+2) and store it where the [b]lue byte used to be.

Lastly we move the [b]lue byte that we stored in the temp variable to the location where the [r]ed byte used to be (i+2), and we close the file with fclose(file).

If everything went ok, the TGA should now be stored in memory as usable OpenGL texture data!

```
for(GLuint i=0; i<int(imageSize); i+=bytesPerPixel) // Loop Through The Image Data
{ // Swaps The 1st And 3rd Bytes ('R'ed and 'B'lue)
temp=texture->imageData[i]; // Temporarily Store The Value At Image Data 'i'
texture->imageData[i] = texture->imageData[i + 2]; // Set The 1st Byte To The Value Of The 3rd
Byte
texture->imageData[i + 2] = temp; // Set The 3rd Byte To The Value In 'temp' (1st Byte Value)
}

fclose (file); // Close The File
```

Now that we have usable data, it's time to make a texture from it. We start off by telling OpenGL we want to create a texture in the memory pointed to by &texture[0].texID.

It's important that you understand a few things before we go on. In the InitGL() code, when we call LoadTGA() we pass it two parameters. The first parameter is &textures[0]. In LoadTGA() we don't make reference to &textures[0]. We make reference to &texture[0] (no 's' at the end). When we modify &texture[0] we are actually modifying textures[0]. texture[0] assumes the identity of textures[0]. I hope that makes sense.

So if we wanted to create a second texture, we would pass the parameter &textures[1]. In LoadTGA() any time we modified texture[0] we would be modifying textures[1]. If we passed &textures[2], texture[0] would assume the identity of &textures[2], etc.

Hard to explain, easy to understand. Of course I wont be happy until I make it really clear :) Last example in english using an example. Say I had a box. I called it box #10. I gave it to my friend and asked him to fill it up. My friend could care less what number it is. To him it's just a box. So he fills what he calls "just a box". He gives it back to me. To me he just filled Box #10 for me. To him he just filled a box. If I give him another box called box #11 and say hey, can you fill this. He'll again think of it as just "box". He'll fill it and give it back to me full. To me he's just filled box #11 for me.

When I give LoadTGA &textures[1] it thinks of it as &texture[0]. It fills it with texture information, and once it's done I am left with a working textures[1]. If I give LoadTGA &textures[2] it again thinks of it as &texture[0]. It fills it with data, and I'm left with a working textures[2]. Make sense :)

Anyways... On to the code! We tell LoadTGA() to build our texture. We bind the texture, and tell OpenGL we want it to be linear filtered.

```
// Build A Texture From The Data
glGenTextures(1, &texture[0].texID); // Generate OpenGL texture IDs

glBindTexture(GL_TEXTURE_2D, texture[0].texID); // Bind Our Texture
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Linear Filtered
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Linear Filtered
```

Now we check to see if the TGA file was 24 or 32 bit. If the TGA was 24 bit, we set the type to GL_RGB. (no alpha channel). If we didn't do this, OpenGL would try to build a texture with an alpha channel. The alpha information wouldn't be there, and the program would probably crash or give an error message.

```
if (texture[0].bpp==24) // Was The TGA 24 Bits
{
type=GL_RGB; // If So Set The 'type' To GL_RGB
}
```

Now we build our texture, the same way we've always done it. But instead of putting the type in ourselves (GL_RGB or GL_RGBA), we substitute the variable type. That way if the program detected that the TGA was 24 bit, the type will be GL_RGB. If our program detected that the TGA was 32 bit, the type would be GL_RGBA.

After the texture has been built, we return true. This lets the InitGL() code know that everything went ok.

```
glTexImage2D(GL_TEXTURE_2D, 0, type, texture[0].width, texture[0].height, 0, type,
GL_UNSIGNED_BYTE, texture[0].imageData);

return true; // Texture Building Went Ok, Return True
}
```

The code below is our standard build a font from a texture code. You've all seen this code before if you've gone through all the tutorials up until now. Nothing really new here, but I figured I'd include the code to make following through the program a little easier.

Only real difference is that I bind to textures[0].texID. Which points to the font texture. Only real difference is that .texID has been added.

```
GLvoid BuildFont(GLvoid) // Build Our Font Display List
{
base=glGenLists(256); // Creating 256 Display Lists
glBindTexture(GL_TEXTURE_2D, textures[0].texID); // Select Our Font Texture
for (int loop1=0; loop1<256; loop1++) // Loop Through All 256 Lists
{
float cx=float(loop1%16)/16.0f; // X Position Of Current Character
float cy=float(loop1/16)/16.0f; // Y Position Of Current Character

glNewList(base+loop1,GL_COMPILE); // Start Building A List
glBegin(GL_QUADS); // Use A Quad For Each Character
glTexCoord2f(cx,1.0f-cy-0.0625f); // Texture Coord (Bottom Left)
glVertex2d(0,16); // Vertex Coord (Bottom Left)
glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f); // Texture Coord (Bottom Right)
glVertex2i(16,16); // Vertex Coord (Bottom Right)
glTexCoord2f(cx+0.0625f,1.0f-cy-0.001f); // Texture Coord (Top Right)
glVertex2i(16,0); // Vertex Coord (Top Right)
glTexCoord2f(cx,1.0f-cy-0.001f); // Texture Coord (Top Left)
glVertex2i(0,0); // Vertex Coord (Top Left)
glEnd(); // Done Building Our Quad (Character)
glTranslated(14,0,0); // Move To The Right Of The Character
glEndList(); // Done Building The Display List
} // Loop Until All 256 Are Built
}
```

KillFont is still the same. We created 256 display lists, so we need to destroy 256 display lists when the program closes.

```
GLvoid KillFont(GLvoid) // Delete The Font From Memory
{
glDeleteLists(base,256); // Delete All 256 Display Lists
}
```

The glPrint() code has only changed a bit. The letters are all stretched on the y axis. Making the letters very tall. I've explained the rest of the code in other tutorials. The stretching is accomplished with the glScalef(x,y,z) command. We leave the ratio at 1.0 on the x axis, we double the size on the y axis (2.0), and we leave it at 1.0 on the z axis.

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...) // Where The Printing Happens
{
char text[1024]; // Holds Our String
va_list ap; // Pointer To List Of Arguments

if (fmt == NULL) // If There's No Text
return; // Do Nothing

va_start(ap, fmt); // Parses The String For Variables
vsprintf(text, fmt, ap); // And Converts Symbols To Actual Numbers
va_end(ap); // Results Are Stored In Text

if (set>1) // Did User Choose An Invalid Character Set?
{
set=1; // If So, Select Set 1 (Italic)
}

glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glLoadIdentity(); // Reset The Modelview Matrix
glTranslated(x,y,0); // Position The Text (0,0 - Top Left)
glListBase(base-32+(128*set)); // Choose The Font Set (0 or 1)

glScalef(1.0f,2.0f,1.0f); // Make The Text 2X Taller

glCallLists(strlen(text),GL_UNSIGNED_BYTE, text); // Write The Text To The Screen
glDisable(GL_TEXTURE_2D); // Disable Texture Mapping
}
```

ReSizeGLScene() sets up an ortho view. Nothing really new. 0,1 is the top left of the screen. 639,480 is the bottom right. This gives us exact screen coordinates in 640 x 480 resolution. Notice that we set the value of swidth to equal the windows current width, and we set the value of sheight to equal the windows current height. Whenever the window is resized or moved, sheight and swidth will be updated.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Window
{
swidth=width; // Set Scissor Width To Window Width
sheight=height; // Set Scissor Height To Window Height
if (height==0) // Prevent A Divide By Zero By
{
height=1; // Making Height Equal One
}
glViewport(0,0,width,height); // Reset The Current Viewport
glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
glLoadIdentity(); // Reset The Projection Matrix
glOrtho(0.0f,640,480,0.0f,-1.0f,1.0f); // Create Ortho 640x480 View (0,0 At Top Left)
glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix
glLoadIdentity(); // Reset The Modelview Matrix
}
```

The init code is very minimal. We load our TGA file. Notice that the first parameter passed is &textures[0]. The second parameter is the name of the file we want to load. In this case, we want to load the Font.TGA file. If LoadTGA() returns false for any reason, the if statement will also return false, causing the program to quit with an "initialization failed" message.

If you wanted to load a second texture you could use the following code: if ((!LoadTGA(&textures[0],"image1.tga")) || (!LoadTGA(&textures[1],"image2.tga"))) { }

After we load the TGA (creating our texture), we build our font, set shading to smooth, set the background color to black, enable clearing of the depth buffer, and select our font texture (bind to it).

Lastly we return true so that our program knows that initialization went ok.

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
if (!LoadTGA(&textures[0],"Data/Font.TGA")) // Load The Font Texture
{
return false; // If Loading Failed, Return False
}

BuildFont(); // Build The Font

glShadeModel(GL_SMOOTH); // Enable Smooth Shading
glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Black Background
glClearDepth(1.0f); // Depth Buffer Setup
glBindTexture(GL_TEXTURE_2D, textures[0].texID); // Select Our Font Texture

return TRUE; // Initialization Went OK
}
```

The draw code is completely new :) we start off by creating a variable of type char called token. Token will hold parsed text later on in the code.

We have another variable called cnt. I use this variable both for counting the number of extensions supported, and for positioning the text on the screen. cnt is reset to zero every time we call DrawGLScene.

We clear the screen and depth buffer and then set the color to bright red (full red intensity, 50% green, 50% blue). at 50 on the x axis and 16 on the y axis we write teh word "Renderer". We also write "Vendor" and "Version" at the top of the screen. The reason each word does not start at 50 on the x axis is because I right justify the words (they all line up on the right side).

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
char *token; // Storage For Our Token
int cnt=0; // Local Counter Variable

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer

glColor3f(1.0f,0.5f,0.5f); // Set Color To Bright Red
glPrint(50,16,1,"Renderer"); // Display Renderer
glPrint(80,48,1,"Vendor"); // Display Vendor Name
glPrint(66,80,1,"Version"); // Display Version
```

Now that we have text on the screen, we change the color to orange, and grab the renderer, vendor name and version number from the video card. We do this by passing GL_RENDERER, GL_VENDOR & GL_VERSION to glGetString(). glGetString will return the requested renderer name, vendor name and version number. The information returned will be text so we need to cast the return information from glGetString as char. All this means is that we tell the program we want the information returned to be characters (text). If you don't include the (char *) you will get an error message. We're printing text, so we need text returned. We grab all three pieces of information and write the information we've grabbed to the right of the previous text.

The information we get from glGetString(GL_RENDERER) will be written beside the red text "Renderer", the information we get from glGetString(GL_VENDOR) will be written to the right of "Vendor", etc.

I'd like to explain casting in more detail, but I'm not really sure of a good way to explain it. If anyone has a good explanation, send it in, and I'll modify my explanation.

After we have the renderer information, vendor information and version number written to the screen, we change the color to a bright blue, and write "NeHe Productions" at the bottom of the screen :) Of course you can change this to anything you want.

```
glColor3f(1.0f,0.7f,0.4f); // Set Color To Orange
glPrint(200,16,1,(char *)glGetString(GL_RENDERER)); // Display Renderer
glPrint(200,48,1,(char *)glGetString(GL_VENDOR)); // Display Vendor Name
glPrint(200,80,1,(char *)glGetString(GL_VERSION)); // Display Version

glColor3f(0.5f,0.5f,1.0f); // Set Color To Bright Blue
glPrint(192,432,1,"NeHe Productions"); // Write NeHe Productions At The Bottom Of The Screen
```

Now we draw a nice white border around the screen, and around the text. We start off by resetting the modelview matrix. Because we've been printing text to the screen, and we might not be at 0,0 on the screen, it's a safe thing to do.

We then set the color to white, and start drawing our borders. A line strip is actually pretty easy to use. You tell OpenGL you want to draw a line strip with glBegin(GL_LINE_STRIP). Then we set the first vertex. Our first vertex will be on the far right side of the screen, and about 63 pixels up from the bottom of the screen (639 on the x axis, 417 on the y axis). Then we set the second vertex. We stay at the same location on the y axis (417), but we move to the far left side of the screen on the x axis (0). A line will be drawn from the right side of the screen (639,417) to the left side of the screen (0,417).

You need to have at least two vertices in order to draw a line (common sense). From the left side of the screen, we move down, right, and then straight up (128 on the y axis).

We then start another line strip, and draw a second box at the top of the screen. If you need to draw ALOT of connected lines, line strips can definitely cut down on the amount of code required as opposed to using regular lines (GL_LINES).

```
glLoadIdentity(); // Reset The ModelView Matrix
glColor3f(1.0f,1.0f,1.0f); // Set The Color To White
glBegin(GL_LINE_STRIP); // Start Drawing Line Strips (Something New)
glVertex2d(639,417); // Top Right Of Bottom Box
glVertex2d( 0,417); // Top Left Of Bottom Box
glVertex2d( 0,480); // Lower Left Of Bottom Box
glVertex2d(639,480); // Lower Right Of Bottom Box
glVertex2d(639,128); // Up To Bottom Right Of Top Box
glEnd(); // Done First Line Strip
glBegin(GL_LINE_STRIP); // Start Drawing Another Line Strip
glVertex2d( 0,128); // Bottom Left Of Top Box
glVertex2d(639,128); // Bottom Right Of Top Box
glVertex2d(639, 1); // Top Right Of Top Box
glVertex2d( 0, 1); // Top Left Of Top Box
glVertex2d( 0,417); // Down To Top Left Of Bottom Box
glEnd(); // Done Second Line Strip
```

Now for something new. A wonderful GL command called glScissor(x,y,w,h). What this command does is creates almost what you would call a window. When GL_SCISSOR_TEST is enabled, the only portion of the screen that you can alter is the portion inside the scissor window. The first line below creates a scissor window starting at 1 on the x axis, and 13.5% (0.135...f) of the way from the bottom of the screen on the y axis. The scissor window will be 638 pixels wide (swidth-2), and 59.7% (0.597...f) of the screen tall.

In the next line we enable scissor testing. Anything we draw OUTSIDE the scissor window will not show up. You could draw a HUGE quad on the screen from 0,0 to 639,480, and you would only see the quad inside the scissor window, the rest of the screen would be unaffected. Very nice command!

The third line of code creates a variable called text that will hold the characters returned by glGetString(GL_EXTENSIONS). malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1) allocates enough memory to hold the entire string returned +1 (so if the string was 50 characters, text would be able to hold all 50 characters).

The next line copies the GL_EXTENSIONS information to text. If we modify the GL_EXTENSIONS information directly, big problems will occur, so instead we copy the information into text, and then manipulate the information stored in text. Basically we're just taking a copy, and storing it in the variable text.

```
glScissor(1 ,int(0.135416f*sheight),swidth-2,int(0.597916f*sheight)); // Define Scissor Region
glEnable(GL_SCISSOR_TEST); // Enable Scissor Testing

char* text=(char*)malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1); // Allocate Memory For
Our Extension String
strcpy (text,(char *)glGetString(GL_EXTENSIONS)); // Grab The Extension List, Store In Text
```

Now for something new. Lets pretend that after grabbing the extension information from the video card, the variable text had the following string of text stored in it... "GL_ARB_multitexture GL_EXT_abgr GL_EXT_bgra". strtok(TextToAnalyze,TextToFind) will scan through the variable text until it finds a " " (space). Once it finds a space, it will copy the text UP TO the space into the variable token. So in our little example, token would be equal to "GL_ARB_multitexture". The space is then replaced with a marker. More about this in a minute.

Next we create a loop that stops once there is no more information left in text. If there is no information in text, token will be equal to nothing (NULL) and the loop will stop.

We increase the counter variable (cnt) by one, and then check to see if the value in cnt is higher than the value of maxtokens. If cnt is higher than maxtokens we make maxtokens equal to cnt. That way if the counter hits 20, maxtokens will also equal 20. It's an easy way to keep track of the maximum value of cnt.

```
token=strtok(text," "); // Parse 'text' For Words, Seperated By " " (spaces)
while(token!=NULL) // While The Token Isn't NULL
{
cnt++; // Increase The Counter
if (cnt>maxtokens) // Is 'maxtokens' Less Than 'cnt'
{
maxtokens=cnt; // If So, Set 'maxtokens' Equal To 'cnt'
}
```

So we have stored the first extension from our list of extensions in the variable token. Next thing to do is set the color to bright green. We then print the variable cnt on the left side of the screen. Notice that we print at 0 on the x axis. This should erase the left (white) border that we drew, but because scissor testing is on, pixels drawn at 0 on the x axis wont be modified. The border can't be drawn over.

The variable is drawn on the far left side of the screen (0 on the x axis). We start drawing at 96 on the y axis. To keep all the text from drawing to the same spot on the screen, we add (cnt*32) to 96. So if we are displaying the first extension, cnt will equal 1, and the text will be drawn at 96+(32*1) (128) on the y axis. If we display the second extension, cnt will equal 2, and the text will be drawn at 96+(32*2) (160) on the y axis.

Notice I also subtract scroll. When the program first runs, scroll will be equal to 0. So our first line of text is drawn at 96+(32*1)-0. If you press the DOWN ARROW, scroll is increased by 2. If scroll was 4, the text would be drawn at 96+(32*1)-4. That means the text would be drawn at 124 instead of 128 on the y axis because of scroll being equal to 4. The top of our scissor window ends at 128 on the y axis. Any part of the text drawn from lines 124-127 on the y axis will not appear on the screen.

Same thing with the bottom of the screen. If cnt was equal to 11 and scroll was equal to 0, the text would be drawn at 96+(32*11)-0 which is 448 on the y axis. Because the scissor window only allows us to draw as far as line 416 on the y axis, the text wouldn't show up at all.

The final result is that we end up with a scrollable window that only allows us to look at 288/32 (9) lines of text. 288 is the height of our scissor window. 32 is the height of the text. By changing the value of scroll we can move the text up or down (offset the text).

The effect is similar to a movie projector. The film rolls by the lens, and all you see is the current frame. You don't see the area above or below the frame. The lens acts as a window similar to the window created by the scissor test.

After we have drawn the current count (cnt) to the screen, we change the color to yellow, move 50 pixels to the right on the x axis, and we write the text stored in the variable token to the screen.

Using our example above, the first line of text displayed on the screen should look like this:

1 GL_ARB_multitexture

```
glColor3f(0.5f,1.0f,0.5f); // Set Color To Bright Green
glPrint(0,96+(cnt*32)-scroll,0,"%i",cnt); // Print Current Extension Number
```

After we have drawn the current count to the screen, we change the color to yellow, move 50 pixels to the right on the x axis, and we write the text stored in the variable token to the screen.

Using our example above, the first line of text displayed on the screen should look like this:

1   GL_ARB_multitexture

```
glColor3f(1.0f,1.0f,0.5f); // Set Color To Yellow
glPrint(50,96+(cnt*32)-scroll,0,token); // Print The Current Token (Parsed Extension Name)
```

After we have displayed the value of token on the screen, we need to check through the variable text to see if any more extensions are supported. Instead of using token=strtok(text," ") like we did above, we replace text with NULL. This tells the command strtok to search from the last marker to the NEXT space in the string of text (text).

In our example above ("GL_ARB_multitexturemarkerGL_EXT_abgr GL_EXT_bgra") there will now be a marker after the text "GL_ARB_multitexture". The line below will start search FROM the marker to the next space. Everything from the marker to the next space will be stored in token. token should end up being "GL_EXT_abgr", and text will end up being "GL_ARB_multitexturemarkerGL_EXT_abgrmarkerGL_EXT_bgra".

Once strtok() has run out of text to store in token, token will become NULL and the loop will stop.

```
token=strtok(NULL," "); // Search For The Next Token
}
```

After all of the extensions have been parsed from the variable text we can disable scissor testing, and free the variable text. This releases the ram we were using to hold the information we got from glGetString(GL_EXTENSIONS).

The next time DrawGLScene() is called, new memory will be allocated. A fresh copy of the information returned by glGetStrings(GL_EXTENSIONS) will be copied into the variable text and the entire process will start over.

```
glDisable(GL_SCISSOR_TEST); // Disable Scissor Testing
```

```
free (text); // Free Allocated Memory
```

The first line below isn't necessary, but I thought it might be a good idea to talk about it, just so everyone knows that it exists. The command glFlush() basically tells OpenGL to finish up what it's doing. If you ever notice flickering in your program (quads disappearing, etc). Try adding the flush command to the end of DrawGLScene. It flushes out the rendering pipeline. You may notice flickering if you're program doesn't have enough time to finish rendering the scene.

Last thing we do is return true to show that everything went ok.

```
glFlush(); // Flush The Rendering Pipeline
return TRUE; // Everything Went OK
}
```

The only thing to note in KillGLWindow() is that I have added KillFont() at the end. That way whenever the window is killed, the font is also killed.

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
{
if (fullscreen) // Are We In Fullscreen Mode?
{
ChangeDisplaySettings(NULL,0); // If So Switch Back To The Desktop
ShowCursor(TRUE); // Show Mouse Pointer
}

if (hRC) // Do We Have A Rendering Context?
{
if (!wglMakeCurrent(NULL,NULL)) // Are We Able To Release The DC And RC Contexts?
{
MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
}

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

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

if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?
{
MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hWnd=NULL; // Set hWnd To NULL
}

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

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

CreateGLWindow(), and WndProc() are the same.

The first change in WinMain() is the title that appears at the top of the window. It should now read "NeHe's Extensions, Scissoring, Token & TGA Loading Tutorial"

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}

// Create Our OpenGL Window
if (!CreateGLWindow("NeHe's Token, Extensions, Scissoring & TGA Loading
Tutorial",640,480,16,fullscreen))
```

```
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit Received?
{
done=TRUE; // ESC or DrawGLScene Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
SwapBuffers(hDC); // Swap Buffers (Double Buffering)

if (keys[VK_F1]) // Is F1 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Token, Extensions, Scissoring & TGA Loading
Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
```

The code below checks to see if the up arrow is being pressed if it is, and scroll is greater than 0, we decrease scroll by 2. This causes the text to move down the screen.

```
if (keys[VK_UP] && (scroll>0)) // Is Up Arrow Being Pressed?
{
scroll-=2; // If So, Decrease 'scroll' Moving Screen Down
}
```

If the down arrow is being pressed and scroll is less than (32*(maxtokens-9)) scroll will be increased by 2, andd the text on the screen will scroll upwards.

32 is the number of lines that each letter takes up. Maxtokens is the total amount of extensions that your video card supports. We subtract 9, because 9 lines can be shown on the screen at once. If we did not subtract 9, we could scroll past the end of the list, causing the list to scroll completely off the screen. Try leaving the -9 out if you're not sure what I mean.

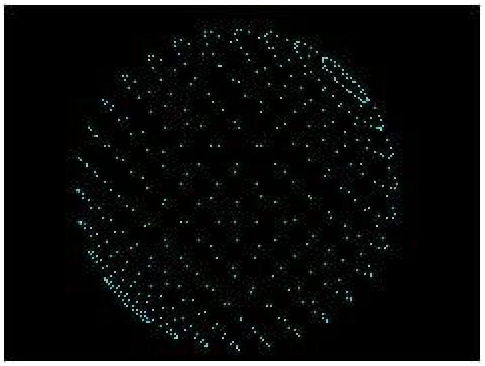
```
if (keys[VK_DOWN] && (scroll<32*(maxtokens-9))) // Is Down Arrow Being Pressed?
{
scroll+=2; // If So, Increase 'scroll' Moving Screen Up
}
}
}
}

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

I hope that you found this tutorial interesting. By the end of this tutorial you should know how to read the vendor name, renderer and version number from your video card. You should also know how to find out what extensions are supported on any video card that supports OpenGL. You should know what scissor testing is, and how it can be used in OpenGL projects of your own, and lastly, you should know how to load TGA Images instead of Bitmap Images for use as textures.

If you find any problems with the tutorial, or you find the information to hard to understand, let me know. I want the tutorials to be the best they can be. Your feedback is important!

Jeff Molofee (**NeHe**)

Lesson 25
Morphing & Loading Objects From A File



Welcome to yet another exciting tutorial! This time we will focus on the effect rather than the graphics, although the final result is pretty cool looking! In this tutorial you will learn how to morph seamlessly from one object to another. Similar to the effect I use in the dolphin demo. Although there are a few catches. First thing to note is that each object must have the same amount of points. Very rare to luck out and get 3 object made up of exactly the same amount of vertices, but it just so happens, in this tutorial we have 3 objects with exactly the same amount of points :) Don't get me wrong, you can use objects with different values, but the transition from one object to another is odd looking and not as smooth.

You will also learn how to read object data from a file. Similar to the format used in lesson 10, although it shouldn't be hard to modify the code to read .ASC files or some other text type data files. In general, it's a really cool effect, a really cool tutorial, so lets begin!

We start off as usual. Including all the required header files, along with the math and standard input / output headers. Notice we don't include glaux. That's because we'll be drawing points rather than textures in this tutorial. After you've got the tutorial figured out, you can try playing with Polygons, Lines, and Textures!

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

HDC hDC=NULL; // Device Context Handle
HGLRC hRC=NULL; // Rendering Context Handle
HWND hWnd=NULL; // Window Handle
HINSTANCE hInstance; // Instance Handle

bool keys[256]; // Key Array
bool active=TRUE; // Program's Active
bool fullscreen=TRUE; // Default Fullscreen To True
```

After setting up all the standard variables, we will add some new variables. xrot, yrot and zrot will hold the current rotation values for the x, y and z axes of the onscreen object. xspeed, yspeed and zspeed will control how fast the object is rotating on each axis. cx, cy and cz control the position of the object on the screen (where it's drawn left to right cx, up and down cy and into and out of the screen cz)

The variable key is a variable that I have included to make sure the user doesn't try to morph from the first shape back into the first shape. This would be pretty pointless and would cause a delay while the points were trying to morph to the position they're already in.

step is a counter variable that counts through all the steps specified by steps. If you increase the value of steps it will take longer for the object to morph, but the movement of the points as they morph will be smoother. Once step is equal to steps we know the morphing has been completed.

The last variable morph lets our program know if it should be morphing the points or leaving them where they are. If it's TRUE, the object is in the process of morphing from one shape to another.

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

```
int key=1; // Used To Make Sure Same Morph Key Is Not Pressed
int step=0,steps=200; // Step Counter And Maximum Number Of Steps
bool morph=FALSE; // Default morph To False (Not Morphing)
```

Now we create a structure to keep track of a vertex. The structure will hold the x, y and z values of any point on the screen. The variables x, y & z are all floating point so we can position the point anywhere on the screen with great accuracy. The structure name is VERTEX.

```
typedef struct // Structure For 3D Points
{
float x, y, z; // X, Y & Z Points
} VERTEX; // Called VERTEX
```

We already have a structure to keep track of vertices, and we know that an object is made up of many vertices so lets create an OBJECT structure. The first variable verts is an integer value that will hold the number of vertices required to make up an object. So if our object has 5 points, the value of verts will be equal to 5. We will set the value later in the code. For now, all you need to know is that verts keeps track of how many points we use to create the object.

The variable points will reference a single VERTEX (x, y and z values). This allows us to grab the x, y or z value of any point using points[{point we want to access}].{x, y or z}.

The name of this structure is... you guessed it... OBJECT!

```
typedef struct // Structure For An Object
{
int verts; // Number Of Vertices For The Object
VERTEX *points; // One Vertice (Vertex x,y & z)
} OBJECT; // Called OBJECT
```

Now that we have created a VERTEX structure and an OBJECT structure we can define some objects.

The variable maxver will be used to keep track of the maximum number of variables used in any of the objects. If one object only had 5 points, another had 20, and the last object had 15, the value of maxver would be equal to the greatest number of points used. So maxver would be equal to 20.

After we define maxver we can define the objects. morph1, morph2, morph3, morph4 & helper are all defined as an OBJECT. *sour & *dest are defined as OBJECT* (pointer to an object). The object is made up of verticies (VERTEX). The first 4 morph{num} objects will hold the 4 objects we want to morph to and from. helper will be used to keep track of changes as the object is morphed. *sour will point to the source object and *dest will point to the object we want to morph to (destination object).

```
int maxver; // Will Eventually Hold The Maximum Number Of Vertices
OBJECT morph1,morph2,morph3,morph4, // Our 4 Morphable Objects (morph1,2,3 & 4)
helper,*sour,*dest; // Helper Object, Source Object, Destination Object
```

Same as always, we declare WndProc().

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

The code below allocates memory for each object, based on the number of vertices we pass to n. *k will point to the object we want to allocate memory for.

The line inside the { }'s allocates the memory for object k's points. A point is an entire VERTEX (3 floats). The memory allocated is the size of VERTEX (3 floats) multiplied by the number of points (n). So if there were 10 points (n=10) we would be allocating room for 30 floating point values (3 floats * 10 points).

```
void objallocate(OBJECT *k,int n) // Allocate Memory For Each Object
{ // And Defines points
k->points=(VERTEX*)malloc(sizeof(VERTEX)*n); // Sets points Equal To VERTEX * Number Of Vertices
} // (3 Points For Each Vertice)
```

The following code frees the object, releasing the memory used to create the object. The object is passed as k. The free command tells our program to release all the points used to make up our object (k).

```
void objfree(OBJECT *k) // Frees The Object (Releasing The Memory)
{
free(k->points); // Frees Points
}
```

The code below reads a string of text from a file. The pointer to our file structure is passed to *f. The variable string will hold the text that we have read in.

We start off be creating a do / while loop. fgets() will read up to 255 characters from our file f and store the characters at *string. If the line read is blank (carriage return \n), the loop will start over, attempting to find a line with text. The while() statement checks for blank lines and if found starts over again.

After the string has been read in we return.

```
void readstr(FILE *f,char *string) // Reads A String From File (f)
{
```

```
do // Do This
{
fgets(string, 255, f); // Gets A String Of 255 Chars Max From f (File)
} while ((string[0] == '/') || (string[0] == '\n')); // Until End Of Line Is Reached
return; // Return
}
```

Now we load in an object. *name points to the filename. *k points to the object we wish to load data into.

We start off with an integer variable called ver. ver will hold the number of vertices used to build the object.

The variables rx, ry & rz will hold the x, y & z values of each vertex.

The variable filein is the pointer to our file structure, and oneline[ ] will be used to hold 255 characters of text.

We open the file name for read in text translated mode (meaning CTRL-Z represents the end of a line). Then we read in a line of text using readstr(filein,oneline). The line of text will be stored in oneline.

After we have read in the text, we scan the line of text (oneline) for the phrase "Vertices: {some number}{carriage return}. If the text is found, the number is stored in the variable ver. This number is the number of vertices used to create the object. If you look at the object text files, you'll see that the first line of text is: Vertices: {some number}.

After we know how many vertices are used we store the results in the objects verts variable. Each object could have a different value if each object had a different number of vertices.

The last thing we do in this section of code is allocate memory for the object. We do this by calling objallocate({object name},{number of verts}).

```
void objload(char *name,OBJECT *k) // Loads Object From File (name)
{
int ver; // Will Hold Vertice Count
float rx,ry,rz; // Hold Vertex X, Y & Z Position
FILE *filein; // Filename To Open
char oneline[255]; // Holds One Line Of Text (255 Chars Max)

filein = fopen(name, "rt"); // Opens The File For Reading Text In Translated Mode
// CTRL Z Symbolizes End Of File In Translated Mode
readstr(filein,oneline); // Jumps To Code That Reads One Line Of Text From The File
sscanf(oneline, "Vertices: %d\n", &ver); // Scans Text For "Vertices: ". Number After Is Stored
In ver
k->verts=ver; // Sets Objects verts Variable To Equal The Value Of ver
objallocate(k,ver); // Jumps To Code That Allocates Ram To Hold The Object
```

We know how many vertices the object has. We have allocated memory, now all that is left to do is read in the vertices. We create a loop using the variable i. The loop will go through all the vertices.

Next we read in a line of text. This will be the first line of valid text underneath the "Vertices: {some number}" line. What we should end up reading is a line with floating point values for x, y & z.

The line is analyzed with sscanf() and the three floating point values are extracted and stored in rx, ry and rz.

```
for (int i=0;i<ver;i++) // Loops Through The Vertices
{
readstr(filein,oneline); // Reads In The Next Line Of Text
sscanf(oneline, "%f %f %f", &rx, &ry, &rz); // Searches For 3 Floating Point Numbers, Store In
rx,ry & rz
```

The following three lines are hard to explain in plain english if you don't understand structures, etc, but I'll try my best :)

The line k->points[i].x=rx can be broken down like this:

rx is the value on the x axis for one of the points.
points[i].x is the x axis position of point[i].
If i is 0 then were are setting the x axis value of point 1, if i is 1, we are setting the x axis value of point 2, and so on.
points[i] is part of our object (which is represented as k).

So if i is equal to 0, what we are saying is: The x axis of point 1 (point[0].x) in our object (k) equals the x axis value we just read from the file (rx).

The other two lines set the y & z axis values for each point in our object.

We loop through all the vertices. If there are not enough vertices, an error might occur, so make sure the text at the beginning of the file "Vertices: {some number}" is actually the number of vertices in the file. Meaning if the top line of the file says "Vertices: 10", there had better be 10 Verticies (x, y and z values)!

After reading in all of the verticies we close the file, and check to see if the variable ver is greater than the variable maxver. If ver is greater than maxver, we set maxver to equal ver. That way if we read in one object and it has 20 verticies, maxver will become 20. If we read in another object, and it has 40 verticies, maxver will become 40. That way we know how many vertices our largest object has.

```
k->points[i].x = rx; // Sets Objects (k) points.x Value To rx
k->points[i].y = ry; // Sets Objects (k) points.y Value To ry
k->points[i].z = rz; // Sets Objects (k) points.z Value To rz
}
fclose(filein); // Close The File

if(ver>maxver) maxver=ver; // If ver Is Greater Than maxver Set maxver Equal To ver
} // Keeps Track Of Highest Number Of Vertices Used
```

The next bit of code may look a little intimidating... it's NOT :) I'll explain it so clearly you'll laugh when you next look at it.

What the code below does is calculates a new position for each point when morphing is enabled. The number of the point to calculate is stored in i. The results will be returned in the VERTEX calculate.

The first variable we create is a VERTEX called a. This will give a an x, y and z value.

Lets look at the first line. The x value of the VERTEX a equals the x value of point[i] (point[i].x) in our SOURCE object minus the x value of point[i] (point[i].x) in our DESTINATION object divided by steps.

So lets plug in some numbers. Lets say our source objects first x value is 40 and our destination objects first x value is 20. We already know that steps is equal to 200! So that means that a.x=(40-20)/200... a.x=(20)/200... a.x=0.1.

What this means is that in order to move from 40 to 20 in 200 steps, we need to move by 0.1 units each calculation. To prove this calculation, multiply 0.1 by 200, and you get 20. 40-20=20 :)

We do the same thing to calculate how many units to move on both the y axis and the z axis for each point. If you increase the value of steps the movements will be even more fine (smooth), but it will take longer to morph from one position to another.

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

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

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

In the code below we set blending for translucency. This allows us to create neat looking trails when the points are moving.

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Set The Blending Function For Translucency
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // This Will Clear The Background Color To Black
glClearDepth(1.0); // Enables Clearing Of The Depth Buffer
glDepthFunc(GL_LESS); // The Type Of Depth Test To Do
glEnable(GL_DEPTH_TEST); // Enables Depth Testing
glShadeModel(GL_SMOOTH); // Enables Smooth Color Shading
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Calculations
```

We set the maxver variable to 0 to start off. We haven't read in any objects so we don't know what the maximum amount of vertices will be.

Next well load in 3 objects. The first object is a sphere. The data for the sphere is stored in the file sphere.txt. The data will be loaded into the object named morph1. We also load a torus, and a tube into objects morph2 and morph3.

```
maxver=0; // Sets Max Vertices To 0 By Default
objload("data/sphere.txt",&morph1); // Load The First Object Into morph1 From File sphere.txt
objload("data/torus.txt",&morph2); // Load The Second Object Into morph2 From File torus.txt
objload("data/tube.txt",&morph3); // Load The Third Object Into morph3 From File tube.txt
```

The 4th object isn't read from a file. It's a bunch of dots randomly scattered around the screen. Because we're not reading the data from a file, we have to manually allocate the memory by calling objallocate(&morph4,468). 468 means we want to allocate enough space to hold 468 vertices (the same amount of vertices the other 3 objects have).

After allocating the space, we create a loop that assigns a random x, y and z value to each point. The random value will be a floating point value from +7 to -7. (14000/1000=14... minus 7 gives us a max value of +7... if the random number is 0, we have a minimum value of 0-7 or -7).

```
objallocate(&morph4,486); // Manually Reserver Ram For A 4th 468 Vertice Object (morph4)
for(int i=0;i<486;i++) // Loop Through All 468 Vertices
{
morph4.points[i].x=((float)(rand()%14000)/1000)-7; // morph4 x Point Becomes A Random Float
Value From -7 to 7
morph4.points[i].y=((float)(rand()%14000)/1000)-7; // morph4 y Point Becomes A Random Float
Value From -7 to 7
morph4.points[i].z=((float)(rand()%14000)/1000)-7; // morph4 z Point Becomes A Random Float
Value From -7 to 7
}
```

We then load the sphere.txt as a helper object. We never want to modify the object data in morph{1/2/3/4} directly. We modify the helper data to make it become one of the 4 shapes. Because we start out displaying morph1 (a sphere) we start the helper out as a sphere as well.

After all of the objects are loaded, we set the source and destination objects (sour and dest) to equal morph1, which is the sphere. This way everything starts out as a sphere.

```
objload("data/sphere.txt",&helper); // Load sphere.txt Object Into Helper (Used As Starting
Point)
sour=dest=&morph1; // Source & Destination Are Set To Equal First Object (morph1)

return TRUE; // Initialization Went OK
}
```

Now for the fun stuff. The actual rendering code :)

We start off normal. Clear the screen, depth buffer and reset the modelview matrix. Then we position the object on the screen using the values stored in cx, cy and cz.

Rotations are done using xrot, yrot and zrot.

The rotation angle is increased based on xpseed, yspeed and zspeed.

Finally 3 temporary variables are created tx, ty and tz, along with a new VERTEX called q.

```
void DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
glTranslatef(cx,cy,cz); // Translate The The Current Position To Start Drawing
glRotatef(xrot,1,0,0); // Rotate On The X Axis By xrot
glRotatef(yrot,0,1,0); // Rotate On The Y Axis By yrot
glRotatef(zrot,0,0,1); // Rotate On The Z Axis By zrot

xrot+=xspeed; yrot+=yspeed; zrot+=zspeed; // Increase xrot,yrot & zrot by xspeed, yspeed &
zspeed

GLfloat tx,ty,tz; // Temp X, Y & Z Variables
VERTEX q; // Holds Returned Calculated Values For One Vertex
```

Now we draw the points and do our calculations if morphing is enabled. glBegin(GL_POINTS) tells OpenGL that each vertex that we specify will be drawn as a point on the screen.

We create a loop to loop through all the vertices. You could use maxver, but because every object has the same number of vertices we'll use morph1.verts.

Inside the loop we check to see if morph is TRUE. If it is we calculate the movement for the current point (i). q.x, q.y and q.z will hold the results. If morph is false, q.x, q.y and q.z will be set to 0 (preventing movement).

the points in the helper object are moved based on the results of we got from calculate(i). (remember earlier that we calculated a point would have to move 0.1 unit to make it from 40 to 20 in 200 steps).

We adjust the each points value on the x, y and z axis by subtracting the number of units to move from helper.

The new helper point is stored in tx, ty and tz. (t{x/y/z}=helper.points[i].{x/y/z}).

```
glBegin(GL_POINTS); // Begin Drawing Points
for(int i=0;i<morph1.verts;i++) // Loop Through All The Verts Of morph1 (All Objects Have
{ // The Same Amount Of Verts For Simplicity, Could Use maxver Also)
if(morph) q=calculate(i); else q.x=q.y=q.z=0; // If morph Is True Calculate Movement Otherwise
Movement=0
helper.points[i].x-=q.x; // Subtract q.x Units From helper.points[i].x (Move On X Axis)
helper.points[i].y-=q.y; // Subtract q.y Units From helper.points[i].y (Move On Y Axis)
helper.points[i].z-=q.z; // Subtract q.z Units From helper.points[i].z (Move On Z Axis)
tx=helper.points[i].x; // Make Temp X Variable Equal To Helper's X Variable
ty=helper.points[i].y; // Make Temp Y Variable Equal To Helper's Y Variable
tz=helper.points[i].z; // Make Temp Z Variable Equal To Helper's Z Variable
```

Now that we have the new position calculated it's time to draw our points. We set the color to a bright bluish color, and the draw the first point with glVertex3f(tx,ty,tz). This draws a point at the newly calculated position.

We then darken the color a little, and move 2 steps in the direction we just calculated instead of one. This moves the point to the newly calculated position, and then moves it again in the same direction. So if it was travelling left at 0.1 units, the next dot would be at 0.2 units. After calculating 2 positions ahead we draw the second point.

Finally we set the color to dark blue, and calculate even further ahead. This time using our example we would move 0.4 units to the left instead of 0.1 or 0.2. The end result is a little tail of particles following as the dots move. With blending, this creates a pretty cool effect!

glEnd() tells OpenGL we are done drawing points.

```
glColor3f(0,1,1); // Set Color To A Bright Shade Of Off Blue
glVertex3f(tx,ty,tz); // Draw A Point At The Current Temp Values (Vertex)
glColor3f(0,0.5f,1); // Darken Color A Bit
tx-=2*q.x; ty-=2*q.y; ty-=2*q.y; // Calculate Two Positions Ahead
glVertex3f(tx,ty,tz); // Draw A Second Point At The Newly Calculate Position
glColor3f(0,0,1); // Set Color To A Very Dark Blue
tx-=2*q.x; ty-=2*q.y; ty-=2*q.y; // Calculate Two More Positions Ahead
glVertex3f(tx,ty,tz); // Draw A Third Point At The Second New Position
} // This Creates A Ghostly Tail As Points Move
glEnd(); // Done Drawing Points
```

The last thing we do is check to see if morph is TRUE and step is less than steps (200). If step is less than 200, we increase step by 1.

If morph is false or step is greater than or equal to steps (200), morph is set to FALSE, the sour (source) object is set to equal the dest (destination) object, and step is set back to 0. This tells the program that morphing is not happening or it has just finished.

```
// If We're Morphing And We Haven't Gone Through All 200 Steps Increase Our Step Counter
// Otherwise Set Morphing To False, Make Source=Destination And Set The Step Counter Back To
Zero.
if(morph && step<=steps)step++; else { morph=FALSE; sour=dest; step=0;}
}
```

The KillGLWindow() code hasn't changed much. The only real difference is that we free all of the objects from memory before we kill the windows. This prevents memory leaks, and is good practice ;)

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
{
objfree(&morph1); // Jump To Code To Release morph1 Allocated Ram
objfree(&morph2); // Jump To Code To Release morph2 Allocated Ram
objfree(&morph3); // Jump To Code To Release morph3 Allocated Ram
objfree(&morph4); // Jump To Code To Release morph4 Allocated Ram
objfree(&helper); // Jump To Code To Release helper Allocated Ram

if (fullscreen) // Are We In Fullscreen Mode?
{
ChangeDisplaySettings(NULL,0); // If So Switch Back To The Desktop
ShowCursor(TRUE); // Show Mouse Pointer
}

if (hRC) // Do We Have A Rendering Context?
{
if (!wglMakeCurrent(NULL,NULL)) // Are We Able To Release The DC And RC Contexts?
{
MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
}

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

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

if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?
{
MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hWnd=NULL; // Set hWnd To NULL
}

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

The CreateGLWindow() and WndProc() code hasn't changed. So I'll skip over it.

```
BOOL CreateGLWindow() // Creates The GL Window
```

```
LRESULT CALLBACK WndProc() // Handle For This Window
```

In WinMain() there are a few changes. First thing to note is the new caption on the title bar :)

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}

// Create Our OpenGL Window
if (!CreateGLWindow("Piotr Cieslak & NeHe's Morphing Points Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if (active && keys[VK_ESCAPE]) // Active? Was There A Quit Received?
{
done=TRUE; // ESC or DrawGLScene Signaled A Quit
}
else // Not Time To Quit, Update Screen
{
DrawGLScene(); // Draw The Scene (Don't Draw When Inactive 1% CPU Use)
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
```

The code below watches for key presses. By now you should understand the code fairly easily. If page up is pressed we increase zspeed. This causes the object to spin faster on the z axis in a positive direction.

If page down is pressed we decrease zspeed. This causes the object to spin faster on the z axis in a negative direction.

If the down arrow is pressed we increase xspeed. This causes the object to spin faster on the x axis in a positive direction.

If the up arrow is pressed we decrease xspeed. This causes the object to spin faster on the x axis in a negative direction.

If the right arrow is pressed we increase yspeed. This causes the object to spin faster on the y axis in a positive direction.

If the left arrow is pressed we decrease yspeed. This causes the object to spin faster on the y axis in a negative direction.

```
if(keys[VK_PRIOR]) // Is Page Up Being Pressed?
zspeed+=0.01f; // Increase zspeed

if(keys[VK_NEXT]) // Is Page Down Being Pressed?
zspeed-=0.01f; // Decrease zspeed

if(keys[VK_DOWN]) // Is Page Up Being Pressed?
xspeed+=0.01f; // Increase xspeed

if(keys[VK_UP]) // Is Page Up Being Pressed?
xspeed-=0.01f; // Decrease xspeed

if(keys[VK_RIGHT]) // Is Page Up Being Pressed?
yspeed+=0.01f; // Increase yspeed

if(keys[VK_LEFT]) // Is Page Up Being Pressed?
yspeed-=0.01f; // Decrease yspeed
```

The following keys physically move the object. 'Q' moves it into the screen, 'Z' moves it towards the viewer, 'W' moves the object up, 'S' moves it down, 'D' moves it right, and 'A' moves it left.

```
if (keys['Q']) // Is Q Key Being Pressed?
cz-=0.01f; // Move Object Away From Viewer

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

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

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

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

if (keys['A']) // Is A Key Being Pressed?
cx-=0.01f; // Move Object Left
```

Now we watch to see if keys 1 through 4 are pressed. If 1 is pressed and key is not equal to 1 (not the current object already) and morph is false (not already in the process of morphing), we set key to 1, so that our program knows we just selected object 1. We then set morph to TRUE, letting our program know it's time to start morphing, and last we set the destination object (dest) to equal object 1 (morph1).

Pressing keys 2, 3, and 4 does the same thing. If 2 is pressed we set dest to morph2, and we set key to equal 2. Pressing 3, sets dest to morph3 and key to 3.

By setting key to the value of the key we just pressed on the keyboard, we prevent the user from trying to morph from a sphere to a sphere or a cone to a cone!

```
if (keys['1'] && (key!=1) && !morph) // Is 1 Pressed, key Not Equal To 1 And Morph False?
{
key=1; // Sets key To 1 (To Prevent Pressing 1 2x In A Row)
morph=TRUE; // Set morph To True (Starts Morphing Process)
dest=&morph1; // Destination Object To Morph To Becomes morph1
}
if (keys['2'] && (key!=2) && !morph) // Is 2 Pressed, key Not Equal To 2 And Morph False?
{
key=2; // Sets key To 2 (To Prevent Pressing 2 2x In A Row)
morph=TRUE; // Set morph To True (Starts Morphing Process)
dest=&morph2; // Destination Object To Morph To Becomes morph2
}
if (keys['3'] && (key!=3) && !morph) // Is 3 Pressed, key Not Equal To 3 And Morph False?
{
key=3; // Sets key To 3 (To Prevent Pressing 3 2x In A Row)
morph=TRUE; // Set morph To True (Starts Morphing Process)
dest=&morph3; // Destination Object To Morph To Becomes morph3
}
if (keys['4'] && (key!=4) && !morph) // Is 4 Pressed, key Not Equal To 4 And Morph False?
{
key=4; // Sets key To 4 (To Prevent Pressing 4 2x In A Row)
morph=TRUE; // Set morph To True (Starts Morphing Process)
dest=&morph4; // Destination Object To Morph To Becomes morph4
}
```

Finally we watch to see if F1 is pressed if it is we toggle from Fullscreen to Windowed mode or Windowed mode to Fullscreen mode!

```
if (keys[VK_F1]) // Is F1 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("Piotr Cieslak & NeHe's Morphing Points Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
}
}
}

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

I hope you have enjoyed this tutorial. Although it's not an incredibly complex tutorial, you can learn alot from the code! The animation in my dolphin demo is done in a similar way to the morphing in this demo. By playing around with the code you can come up with some really cool effects. Dots turning into words. Faked animation, and more! You may even want to try using solid polygons or lines instead of dots. The effect can be quite impressive!

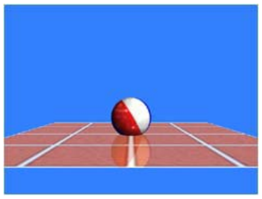
Piotr's code is new and refreshing. I hope that after reading through this tutorial you have a better understanding on how to store and load object data from a file, and how to manipulate the data to create cool GL effects in your own programs! The .html for this tutorial took 3 days to write. If you notice any mistakes please let me know. Alot of it was written late at night, meaning a few mistakes may have crept in. I want these tutorials to be the best they can be. Feedback is appreciated!

RabidHaMsTeR released a demo called "Morph" before this tutorial was written that shows off a more advanced version of this effect. You can check it out yourself at http://homepage.ntlworld.com/fj.williams/PgSoftware.html.

Piotr Cieslak

Jeff Molofee (**NeHe**)

Lesson 26
Clipping & Reflections Using The Stencil Buffer



Welcome to another exciting tutorial. The code for this tutorial was written by Banu Cosmin. The tutorial was of course written by myself (NeHe). In this tutorial you will learn how to create EXTREMELY realistic reflections. Nothing fake here! The objects being reflected will not show up underneath the floor or on the other side of a wall. True reflections!

A very important thing to note about this tutorial: Because the Voodoo 1, 2 and some other cards do not support the stencil buffer, this demo will NOT run on those cards. It will ONLY run on cards that support the stencil buffer. If you're not sure if your card supports the stencil buffer, download the code, and try running the demo. Also, this demo requires a fairly decent processor and graphics card. Even on my GeForce I notice there is a little slow down at times. This demo runs best in 32 bit color mode!

As video cards get better, and processors get faster, I can see the stencil buffer becoming more popular. If you have the hardware and you're ready to reflect, read on!

The first part of the code is fairly standard. We include all necessary header files, and set up our Device Context, Rendering Context, etc.

```
#include <windows.h> // Header File For Windows
#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
#include <stdio.h> // Header File For Standard Input / Output

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

Next we have the standard variables to keep track of key presses (keys[ ]), whether or not the program is active (active), and if we should use fullscreen mode or windowed mode (fullscreen).

```
bool keys[256]; // Array Used For The Keyboard Routine
bool active=TRUE; // Window Active Flag Set To TRUE By Default
bool fullscreen=TRUE; // Fullscreen Flag Set To Fullscreen Mode By Default
```

Next we set up our lighting variables. LightAmb[ ] will set our ambient light. We will use 70% red, 70% green and 70% blue, creating a light that is 70% bright white. LightDif[ ] will set the diffuse lighting (the amount of light evenly reflected off the surface of our object). In this case we want to reflect full intensity light. Lastly we have LightPos[ ] which will be used to position our light. In this case we want the light 4 units to the right, 4 units up, and 6 units towards the viewer. If we could actually see the light, it would be floating in front of the top right corner of our screen.

```
// Light Parameters
static GLfloat LightAmb[] = {0.7f, 0.7f, 0.7f, 1.0f}; // Ambient Light
static GLfloat LightDif[] = {1.0f, 1.0f, 1.0f, 1.0f}; // Diffuse Light
static GLfloat LightPos[] = {4.0f, 4.0f, 6.0f, 1.0f}; // Light Position
```

We set up a variable called q for our quadratic object, xrot and yrot to keep track of rotation. xrotspeed and yrotspeed control the speed our object rotates at. zoom is used to zoom in and out of the scene (we start at -7 which shows us the entire scene) and height is the height of the ball above the floor.

We then make room for our 3 textures with texture[3], and define WndProc().

```
GLUquadricObj *q; // Quadratic For Drawing A Sphere

GLfloat xrot = 0.0f; // X Rotation
GLfloat yrot = 0.0f; // Y Rotation
GLfloat xrotspeed = 0.0f; // X Rotation Speed
GLfloat yrotspeed = 0.0f; // Y Rotation Speed
GLfloat zoom = -7.0f; // Depth Into The Screen
GLfloat height = 2.0f; // Height Of Ball From Floor

GLuint texture[3]; // 3 Textures

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

The ReSizeGLScene() and LoadBMP() code has not changed so I will skip over both sections of code.

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

AUX_RGBImageRec *LoadBMP(char *Filename) // Loads A Bitmap Image
```

The load texture code is pretty standard. You've used it many times before in the previous tutorials. We make room for 3 textures, then we load the three images, and create linear filtered textures from the image data. The bitmap files we use are located in the DATA directory.

```
int LoadGLTextures() // Load Bitmaps And Convert To Textures
{
int Status=FALSE; // Status Indicator
AUX_RGBImageRec *TextureImage[3]; // Create Storage Space For The Textures
memset(TextureImage,0,sizeof(void *)*3); // Set The Pointer To NULL
if ((TextureImage[0]=LoadBMP("Data/EnvWall.bmp")) && // Load The Floor Texture
(TextureImage[1]=LoadBMP("Data/Ball.bmp")) && // Load the Light Texture
(TextureImage[2]=LoadBMP("Data/EnvRoll.bmp"))) // Load the Wall Texture
{
Status=TRUE; // Set The Status To TRUE
glGenTextures(3, &texture[0]); // Create The Texture
for (int loop=0; loop<3; loop++) // Loop Through 5 Textures
{
glBindTexture(GL_TEXTURE_2D, texture[loop]);
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
}
for (loop=0; loop<3; loop++) // Loop Through 5 Textures
{
if (TextureImage[loop]) // If Texture Exists
{
if (TextureImage[loop]->data) // If Texture Image Exists
{
free(TextureImage[loop]->data); // Free The Texture Image Memory
}
free(TextureImage[loop]); // Free The Image Structure
}
}
}
return Status; // Return The Status
}
```

A new command called glClearStencil is introduced in the init code. Passing 0 as a parameter tells OpenGL to disable clearing of the stencil buffer. You should be familiar with the rest of the code by now. We load our textures and enable smooth shading. The clear color is set to an off blue and the clear depth is set to 1.0f. The stencil clear value is set to 0. We enable depth testing, and set the depth test value to less than or equal to. Our perspective correction is set to nicest (very good quality) and 2d texture mapping is enabled.

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
if (!LoadGLTextures()) // If Loading The Textures Failed
{
return FALSE; // Return False
}
glShadeModel(GL_SMOOTH); // Enable Smooth Shading
glClearColor(0.2f, 0.5f, 1.0f, 1.0f); // Background
glClearDepth(1.0f); // Depth Buffer Setup
glClearStencil(0); // Clear The Stencil Buffer To 0
glEnable(GL_DEPTH_TEST); // Enables Depth Testing
glDepthFunc(GL_LEQUAL); // The Type Of Depth Testing To Do
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Calculations
glEnable(GL_TEXTURE_2D); // Enable 2D Texture Mapping
```

Now it's time to set up light 0. The first line below tells OpenGL to use the values stored in LightAmb for the Ambient light. If you remember at the beginning of the code, the rgb values of LightAmb were all 0.7f, giving us a white light at 70% full intensity. We then set the Diffuse light using the values stored in LightDif and position the light using the x,y,z values stored in LightPos.

After we have set the light up we can enable it with glEnable(GL_LIGHT0). Even though the light is enabled, you will not see it until we enable lighting with the last line of code.

Note: If we wanted to turn off all lights in a scene we would use glDisable(GL_LIGHTING). If we wanted to disable just one of our lights we would use glDisable(GL_LIGHT{0-7}). This gives us alot of control over the lighting and what lights are on and off. Just remember if GL_LIGHTING is disabled, you will not see lights!

```
glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmb); // Set The Ambient Lighting For Light0
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDif); // Set The Diffuse Lighting For Light0
glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Set The Position For Light0

glEnable(GL_LIGHT0); // Enable Light 0
glEnable(GL_LIGHTING); // Enable Lighting
```

In the first line below, we create a new quadratic object. The second line tells OpenGL to generate smooth normals for our quadratic object, and the third line tells OpenGL to generate texture coordinates for our quadratic. Without the second and third lines of code, our object would use flat shading and we wouldn't be able to texture it.

The fourth and fifth lines tell OpenGL to use the Sphere Mapping algorithm to generate the texture coordinates. This allows us to sphere map the quadratic object.

```
q = gluNewQuadric(); // Create A New Quadratic
gluQuadricNormals(q, GL_SMOOTH); // Generate Smooth Normals For The Quad
gluQuadricTexture(q, GL_TRUE); // Enable Texture Coords For The Quad

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set Up Sphere Mapping
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set Up Sphere Mapping

return TRUE; // Initialization Went OK
}
```

The code below will draw our object (which is a cool looking environment mapped beach ball).

We set the color to full intensity white and bind to our BALL texture (the ball texture is a series of red, white and blue stripes).

After selecting our texture, we draw a Quadratic Sphere with a radius of 0.35f, 32 slices and 16 stacks (up and down).

```
void DrawObject() // Draw Our Ball
{
glColor3f(1.0f, 1.0f, 1.0f); // Set Color To White
glBindTexture(GL_TEXTURE_2D, texture[1]); // Select Texture 2 (1)
gluSphere(q, 0.35f, 32, 16); // Draw First Sphere
```

After drawing the first sphere, we select a new texture (EnvRoll), set the alpha value to 40% and enable blending based on the source alpha value. glEnable(GL_TEXTURE_GEN_S) and glEnable(GL_TEXTURE_GEN_T) enables sphere mapping.

After doing all that, we redraw the sphere, disable sphere mapping and disable blending.

The final result is a reflection that almost looks like bright points of light mapped to the beach ball. Because we enable sphere mapping, the texture is always facing the viewer, even as the ball spins. We blend so that the new texture doesn't cancel out the old texture (a form of multitexturing).

```
glBindTexture(GL_TEXTURE_2D, texture[2]); // Select Texture 3 (2)
glColor4f(1.0f, 1.0f, 1.0f, 0.4f); // Set Color To White With 40% Alpha
glEnable(GL_BLEND); // Enable Blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Set Blending Mode To Mix Based On SRC Alpha
glEnable(GL_TEXTURE_GEN_S); // Enable Sphere Mapping
glEnable(GL_TEXTURE_GEN_T); // Enable Sphere Mapping

gluSphere(q, 0.35f, 32, 16); // Draw Another Sphere Using New Texture
// Textures Will Mix Creating A MultiTexture Effect (Reflection)
glDisable(GL_TEXTURE_GEN_S); // Disable Sphere Mapping
glDisable(GL_TEXTURE_GEN_T); // Disable Sphere Mapping
glDisable(GL_BLEND); // Disable Blending
}
```

The code below draws the floor that our ball hovers over. We select the floor texture (EnvWall), and draw a single texture mapped quad on the z-axis. Pretty simple!

```
void DrawFloor() // Draws The Floor
{
glBindTexture(GL_TEXTURE_2D, texture[0]); // Select Texture 1 (0)
glBegin(GL_QUADS); // Begin Drawing A Quad
glNormal3f(0.0, 1.0, 0.0); // Normal Pointing Up
glTexCoord2f(0.0f, 1.0f); // Bottom Left Of Texture
glVertex3f(-2.0, 0.0, 2.0); // Bottom Left Corner Of Floor

glTexCoord2f(0.0f, 0.0f); // Top Left Of Texture
glVertex3f(-2.0, 0.0,-2.0); // Top Left Corner Of Floor

glTexCoord2f(1.0f, 0.0f); // Top Right Of Texture
glVertex3f( 2.0, 0.0,-2.0); // Top Right Corner Of Floor

glTexCoord2f(1.0f, 1.0f); // Bottom Right Of Texture
glVertex3f( 2.0, 0.0, 2.0); // Bottom Right Corner Of Floor
glEnd(); // Done Drawing The Quad
}
```

Now for the fun stuff. Here's where we combine all the objects and images to create our reflective scene.

We start off by clearing the screen (GL_COLOR_BUFFER_BIT) to our default clear color (off blue). The depth (GL_DEPTH_BUFFER_BIT) and stencil (GL_STENCIL_BUFFER_BIT) buffers are also cleared. Make sure you include the stencil buffer code, it's new and easy to overlook! It's important to note when we clear the stencil buffer, we are filling it with 0's.

After clearing the screen and buffers, we define our clipping plane equation. The plane equation is used for clipping the reflected image.

The equation eqr[]={0.0f,-1.0f, 0.0f, 0.0f} will be used when we draw the reflected image. As you can see, the value for the y-plane is a negative value. Meaning we will only see pixels if they are drawn below the floor or at a negative value on the y-axis. Anything drawn above the floor will not show up when using this equation.

More on clipping later... read on.

```
int DrawGLScene(GLvoid) // Draw Everything
{
// Clear Screen, Depth Buffer & Stencil Buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

// Clip Plane Equations
double eqr[] = {0.0f,-1.0f, 0.0f, 0.0f}; // Plane Equation To Use For The Reflected Objects
```

So we have cleared the screen, and defined our clipping planes. Now for the fun stuff!

We start off by resetting the modelview matrix. Which of course starts all drawing in the center of the screen. We then translate down 0.6f units (to add a small perspective tilt to the floor) and into the screen based on the value of zoom. To better explain why we translate down 0.6f units, I'll explain using a simple example. If you were looking at the side of a piece of paper at exactly eye level, you would barely be able to see it. It would more than likely look like a thin line. If you moved the paper down a little, it would no longer look like a line. You would see more of the paper, because your eyes would be looking down at the page instead of directly at the edge of the paper.

```
glLoadIdentity(); // Reset The Modelview Matrix
glTranslatef(0.0f, -0.6f, zoom); // Zoom And Raise Camera Above The Floor (Up 0.6 Units)
```

Next we set the color mask. Something new to this tutorial! The 4 values for color mask represent red, green, blue and alpha. By default all the values are set to GL_TRUE.

If the red value of glColorMask({red},{green},{blue},{alpha}) was set to GL_TRUE, and all of the other values were 0 (GL_FALSE), the only color that would show up on the screen is red. If the value for red was 0 (GL_FALSE), but the other values were all GL_TRUE, every color except red would be drawn to the screen.

We don't want anything drawn to the screen at the moment, with all of the values set to 0 (GL_FALSE), colors will not be drawn to the screen.

```
glColorMask(0,0,0,0); // Set Color Mask
```

Now even more fun stuff... Setting up the stencil buffer and stencil testing!

We start off by enabling stencil testing. Once stencil testing has been enabled, we are able to modify the stencil buffer.

It's very hard to explain the commands below so please bear with me, and if you have a better explanation, please let me know. In the code below we set up a test. The line glStencilFunc(GL_ALWAYS, 1, 1) tells OpenGL what type of test we want to do on each pixel when an object is drawn to the screen.

GL_ALWAYS just tells OpenGL the test will always pass. The second parameter (1) is a reference value that we will test in the third line of code, and the third parameter is a mask. The mask is a value that is ANDed with the reference value and stored in the stencil buffer when the test is done. A reference value of 1 ANDed with a mask value of 1 is 1. So if the test goes well and we tell OpenGL to, it will place a one in the stencil buffer (reference&mask=1).

Quick note: Stencil testing is a per pixel test done each time an object is drawn to the screen. The reference value ANDed with the mask value is tested against the current stencil value ANDed with the mask value.

The third line of code tests for three different conditions based on the stencil function we decided to use. The first two parameters are GL_KEEP, and the third is GL_REPLACE.

The first parameter tells OpenGL what to do if the test fails. Because the first parameter is GL_KEEP, if the test fails (which it can't because we have the funtion set to GL_ALWAYS), we would leave the stencil value set at whatever it currently is.

The second parameter tells OpenGL what do do if the stencil test passes, but the depth test fails. In the code below, we eventually disable depth testing so this parameter can be ignored.

The third parameter is the important one. It tells OpenGL what to do if the test passes! In our code we tell OpenGL to replace (GL_REPLACE) the value in the stencil buffer. The value we put into the stencil buffer is our reference value ANDed with our mask value which is 1.

After setting up the type of testing we want to do, we disable depth testing and jump to the code that draws our floor.

In simple english I will try to sum up everything that the code does up until now...

We tell OpenGL not to draw any colors to the screen. This means that when we draw the floor, it wont show up on the screen. BUT... each spot on the screen where the object (our floor) should be if we could see it will be tested based on the type of stencil testing we decide to do. The stencil buffer starts out full of 0's (empty). We want to set the stencil value to 1 wherever our object would have been drawn if we could see it. So we tell OpenGL we don't care about testing. If a pixel should have been drawn to the screen, we want that spot marked with a 1. GL_ALWAYS does exactly that. Our reference and mask values of 1 make sure that the value placed into the stencil buffer is indeed going to be 1! As we invisibly draw, our stencil operation checks each pixel location, and replaces the 0 with a 1.

```
glEnable(GL_STENCIL_TEST); // Enable Stencil Buffer For "marking" The Floor
glStencilFunc(GL_ALWAYS, 1, 1); // Always Passes, 1 Bit Plane, 1 As Mask
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE); // We Set The Stencil Buffer To 1 Where We Draw Any
Polygon
// Keep If Test Fails, Keep If Test Passes But Buffer Test Fails
// Replace If Test Passes
glDisable(GL_DEPTH_TEST); // Disable Depth Testing
DrawFloor(); // Draw The Floor (Draws To The Stencil Buffer)
// We Only Want To Mark It In The Stencil Buffer
```

So now we have an invisible stencil mask of the floor. As long as stencil testing is enabled, the only places pixels will show up are places where the stencil buffer has a value of 1. All of the pixels on the screen where the invisible floor was drawn will have a stencil value of 1. Meaning as long as stencil testing is enabled, the only pixels that we will see are the pixels that we draw in the same spot our invisible floor was defined in the stencil buffer. The trick behind creating a real looking reflection that reflects in the floor and nowhere else!

So now that we know the ball reflection will only be drawn where the floor should be, it's time to draw the reflection! We enable depth testing, and set the color mask back to all ones (meaning all the colors will be drawn to the screen).

Instead of using GL_ALWAYS for our stencil function we are going to use GL_EQUAL. We'll leave the reference and mask values at 1. For the stencil operation we will set all the parameters to GL_KEEP. In english, any object we draw this time around will actually appear on the screen (because the color mask is set to true for each color). As long as stencil testing is enabled pixels will ONLY be drawn if the stencil buffer has a value of 1 (reference value ANDed with the mask, which is 1 EQUALS (GL_EQUAL) the stencil buffer value ANDed with the mask, which is also 1). If the stencil value is not 1 where the current pixel is being drawn it will not show up! GL_KEEP just tells OpenGL not to modify any values in the stencil buffer if the test passes OR fails!

```
glEnable(GL_DEPTH_TEST); // Enable Depth Testing
glColorMask(1,1,1,1); // Set Color Mask to TRUE, TRUE, TRUE, TRUE
glStencilFunc(GL_EQUAL, 1, 1); // We Draw Only Where The Stencil Is 1
// (I.E. Where The Floor Was Drawn)
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); // Don't Change The Stencil Buffer
```

Now we enable the mirrored clipping plane. This plane is defined by eqr, and only allows object to be drawn from the center of the screen (where the floor is) down to the bottom of the screen (any negative value on the y-axis). That way the reflected ball that we draw can't come up through the center of the floor. That would look pretty bad if it did. If you don't understand what I mean, remove the first line below from the source code, and move the real ball (non reflected) through the floor. If clipping is not enabled, you will see the reflected ball pop out of the floor as the real ball goes into the floor.

After we enable clipping plane0 (usually you can have from 0-5 clipping planes), we define the plane by telling it to use the parameters stored in eqr.

We push the matrix (which basically saves the position of everything on the screen) and use glScalef(1.0f,-1.0f,1.0f) to flip the object upside down (creating a real looking reflection). Setting the y value of glScalef({x},{y},{z}) to a negative value forces OpenGL to render opposite on the y-axis. It's almost like flipping the entire screen upside down. When position an object at a positive value on the y-axis, it will appear at the bottom of the screen instead of at the top. When you rotate an object towards yourself, it will rotate away from you. Everything will be mirrored on the y-axis until you pop the matrix or set the y value back to 1.0f instead of -1.0f using glScalef({x},{y},{z}).

```
glEnable(GL_CLIP_PLANE0); // Enable Clip Plane For Removing Artifacts
// (When The Object Crosses The Floor)
glClipPlane(GL_CLIP_PLANE0, eqr); // Equation For Reflected Objects
glPushMatrix(); // Push The Matrix Onto The Stack
glScalef(1.0f, -1.0f, 1.0f); // Mirror Y Axis
```

The first line below positions our light to the location specified by LightPos. The light should shine on the bottom right of the reflected ball creating a very real looking light source. The position of the light is also mirrored. On the real ball (ball above the floor) the light is positioned at the top right of your screen, and shines on the top right of the real ball. When drawing the reflected ball, the light is positioned at the bottom right of your screen.

We then move up or down on the y-axis to the value specified by height. Translations are mirrored, so if the value of height is 5.0f, the position we translate to will be mirrored (-5.0f). Positioning the reflected image under the floor, instead of above the floor!

After position our reflected ball, we rotate the ball on both the x axis and y axis, based on the values of xrot and yrot. Keep in mind that any rotations on the x axis will also be mirrored. So if the real ball (ball above the floor) is rolling towards you on the x-axis, it will be rolling away from you in the reflection.

After positioning the reflected ball and doing our rotations we draw the ball by calling DrawObject(), and pop the matrix (restoring things to how they were before we drew the ball). Popping the matrix all cancels mirroring on the y-axis.

We then disable our clipping plane (plane0) so that we are not stuck drawing only to the bottom half of the screen, and last, we disable stencil testing so that we can draw to other spots on the screen other than where the floor should be.

Note that we draw the reflected ball before we draw the floor. I'll explain why later on.

```
glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Set Up Light0
glTranslatef(0.0f, height, 0.0f); // Position The Object
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Rotate Local Coordinate System On X Axis
glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Rotate Local Coordinate System On Y Axis
DrawObject(); // Draw The Sphere (Reflection)
glPopMatrix(); // Pop The Matrix Off The Stack
glDisable(GL_CLIP_PLANE0); // Disable Clip Plane For Drawing The Floor
glDisable(GL_STENCIL_TEST); // We Don't Need The Stencil Buffer Any More (Disable)
```

We start off this section of code by positioning our light. The y-axis is no longer being mirrored so drawing the light this time around will position it at the top of the screen instead of the bottom right of the screen.

We enable blending, disable lighting, and set the alpha value to 80% using the command glColor4f(1.0f,1.0f,1.0f,0.8f). The blending mode is set up using glBlendFunc(), and the semi transparent floor is drawn over top of the reflected ball.

If we drew the floor first and then the reflected ball, the effect wouldn't look very good. By drawing the ball and then the floor, you can see a small amount of coloring from the floor mixed into the coloring of the ball. If I was looking into a BLUE mirror, I would expect the reflection to look a little blue. By rendering the ball first, the reflected image looks like it's tinted the color of the floor.

```
glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Set Up Light0 Position
glEnable(GL_BLEND); // Enable Blending (Otherwise The Reflected Object Wont Show)
glDisable(GL_LIGHTING); // Since We Use Blending, We Disable Lighting
glColor4f(1.0f, 1.0f, 1.0f, 0.8f); // Set Color To White With 80% Alpha
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Blending Based On Source Alpha And 1 Minus
Dest Alpha
DrawFloor(); // Draw The Floor To The Screen
```

Now we draw the 'real' ball (the one that floats above the floor). We disabled lighting when we drew the floor, but now it's time to draw another ball so we will turn lighting back on.

We don't need blending anymore so we disable blending. If we didn't disable blending, the colors from the floor would mix with the colors of our 'real' ball when it was floating over top of the floor. We don't want the 'real' ball to look like the reflection so we disable blending.

We are not going to clip the actual ball. If the real ball goes through the floor, we should see it come out the bottom. If we were using clipping the ball wouldn't show up after it went through the floor. If you didn't want to see the ball come through the floor, you would set up a clipping equation that set the Y value to +1.0f, then when the ball went through the floor, you wouldn't see it (you would only see the ball when it was drawn on at a positive value on the y-axis. For this demo, there's no reason we shouldn't see it come through the floor.

We then translate up or down on the y-axis to the position specified by height. Only this time the y-axis is not mirrored, so the ball travels the opposite direction that the reflected image travels. If we move the 'real' ball down the reflected ball will move up. If we move the 'real' ball up, the reflected ball will move down.

We rotate the 'real' ball, and again, because the y-axis is not mirrored, the ball will spin the opposite direction of the reflected ball. If the reflected ball is rolling towards you the 'real' ball will be rolling away from you. This creates the illusion of a real reflection.

After positioning and rotating the ball, we draw the 'real' ball by calling DrawObject().

```
glEnable(GL_LIGHTING); // Enable Lighting
glDisable(GL_BLEND); // Disable Blending
glTranslatef(0.0f, height, 0.0f); // Position The Ball At Proper Height
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Rotate On The X Axis
glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Rotate On The Y Axis
DrawObject(); // Draw The Ball
```

The following code rotates the ball on the x and y axis. By increasing xrot by xrotspeed we rotate the ball on the x-axis. By increasing yrot by yrotspeed we spin the ball on the y-axis. If xrotspeed is a very high value in the positive or negative direction the ball will spin quicker than if xrotspeed was a low value, closer to 0.0f. Same goes for yrotspeed. The higher the value, the faster the ball spins on the y-axis.

Before we return TRUE, we do a glFlush(). This tells OpenGL to render everything left in the GL pipeline before continuing, and can help prevent flickering on slower video cards.

```
xrot += xrotspeed; // Update X Rotation Angle By xrotspeed
yrot += yrotspeed; // Update Y Rotation Angle By yrotspeed
glFlush(); // Flush The GL Pipeline
return TRUE; // Everything Went OK
}
```

The following code will watch for key presses. The first 4 lines check to see if you are pressing one of the 4 arrow keys. If you are, the ball is spun right, left, down or up.

The next 2 lines check to see if you are pressing the 'A' or 'Z' keys. Pressing 'A' will zoom you in closer to the ball and pressing 'Z' will zoom you away from the ball.

Pressing 'PAGE UP' will increase the value of height moving the ball up, and pressing 'PAGE DOWN' will decrease the value of height moving the ball down (closer to the floor).

```
void ProcessKeyboard() // Process Keyboard Results
{
if (keys[VK_RIGHT]) yrotspeed += 0.08f; // Right Arrow Pressed (Increase yrotspeed)
if (keys[VK_LEFT]) yrotspeed -= 0.08f; // Left Arrow Pressed (Decrease yrotspeed)
if (keys[VK_DOWN]) xrotspeed += 0.08f; // Down Arrow Pressed (Increase xrotspeed)
if (keys[VK_UP]) xrotspeed -= 0.08f; // Up Arrow Pressed (Decrease xrotspeed)

if (keys['A']) zoom +=0.05f; // 'A' Key Pressed ... Zoom In
if (keys['Z']) zoom -=0.05f; // 'Z' Key Pressed ... Zoom Out

if (keys[VK_PRIOR]) height +=0.03f; // Page Up Key Pressed Move Ball Up
if (keys[VK_NEXT]) height -=0.03f; // Page Down Key Pressed Move Ball Down
}
```

The KillGLWindow() code hasn't changed, so I'll skip over it.

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
```

You can skim through the following code. Even though only one line of code has changed in CreateGLWindow(), I have included all of the code so it's easier to follow through the tutorial.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
GLuint PixelFormat; // Holds The Results After Searching For A Match
WNDCLASS wc; // Windows Class Structure
DWORD dwExStyle; // Window Extended Style
DWORD dwStyle; // Window Style

fullscreen=fullscreenflag; // Set The Global Fullscreen Flag

hInstance = GetModuleHandle(NULL); // Grab An Instance For Our Window
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // Redraw On Size, And Own DC For Window
wc.lpfnWndProc = (WNDPROC) WndProc; // WndProc Handles Messages
wc.cbClsExtra = 0; // No Extra Window Data
wc.cbWndExtra = 0; // No Extra Window Data
wc.hInstance = hInstance; // Set The Instance
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Load The Default Icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Load The Arrow Pointer
wc.hbrBackground = NULL; // No Background Required For GL
wc.lpszMenuName = NULL; // We Don't Want A Menu
wc.lpszClassName = "OpenGL"; // Set The Class Name

if (!RegisterClass(&wc)) // Attempt To Register The Window Class
{
MessageBox(NULL,"Failed To Register The Window Class.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

if (fullscreen) // Attempt Fullscreen Mode?
{
DEVMODE dmScreenSettings; // Device Mode
memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's Cleared
dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode Structure
dmScreenSettings.dmPelsWidth = width; // Selected Screen Width
dmScreenSettings.dmPelsHeight = height; // Selected Screen Height
dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pixel
dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

// Try To Set Selected Mode And Get Results. NOTE: CDS_FULLSCREEN Gets Rid Of Start Bar
if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
{
// If The Mode Fails, Offer Two Options. Quit Or Use Windowed Mode
if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use
Windowed Mode Instead?","NeHe GL",MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
{
fullscreen=FALSE; // Windowed Mode Selected. Fullscreen = FALSE
}
else
{
// Pop Up A Message Box Letting User Know The Program Is Closing
MessageBox(NULL,"Program Will Now Close.","ERROR",MB_OK|MB_ICONSTOP);
return FALSE; // Return FALSE
}
}
}

if (fullscreen) // Are We Still In Fullscreen Mode?
{
```

```
dwExStyle=WS_EX_APPWINDOW; // Window Extended Style
dwStyle=WS_POPUP | WS_CLIPSIBLINGS | WS_CLIPCHILDREN; // Windows Style
ShowCursor(FALSE); // Hide Mouse Pointer
}
else
{
dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // Window Extended Style
dwStyle=WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;// Windows Style
}

// Create The Window
if (!(hWnd=CreateWindowEx( dwExStyle, // Extended Style For The Window
"OpenGL", // Class Name
title, // Window Title
dwStyle, // Window Style
0, 0, // Window Position
width, height, // Selected Width And Height
NULL, // No Parent Window
NULL, // No Menu
hInstance, // Instance
NULL))) // Dont Pass Anything To WM_CREATE
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Window Creation Error.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

static PIXELFORMATDESCRIPTOR pfd= // pfd Tells Windows How We Want Things To Be
{
sizeof(PIXELFORMATDESCRIPTOR), // Size Of This Pixel Format Descriptor
1, // Version Number
PFD_DRAW_TO_WINDOW | // Format Must Support Window
PFD_SUPPORT_OPENGL | // Format Must Support OpenGL
PFD_DOUBLEBUFFER, // Must Support Double Buffering
PFD_TYPE_RGBA, // Request An RGBA Format
bits, // Select Our Color Depth
0, 0, 0, 0, 0, 0, // Color Bits Ignored
0, // No Alpha Buffer
0, // Shift Bit Ignored
0, // No Accumulation Buffer
0, 0, 0, 0, // Accumulation Bits Ignored
16, // 16Bit Z-Buffer (Depth Buffer)
```

The only change in this section of code is the line below. It is *VERY IMPORTANT* you change the value from 0 to 1 or some other non zero value. In all of the previous tutorials the value of the line below was 0. In order to use Stencil Buffering this value HAS to be greater than or equal to 1. This value is the number of bits you want to use for the stencil buffer.

```
1, // Use Stencil Buffer ( * Important * )
0, // No Auxiliary Buffer
PFD_MAIN_PLANE, // Main Drawing Layer
0, // Reserved
0, 0, 0 // Layer Masks Ignored
};

if (!(hDC=GetDC(hWnd))) // Did We Get A Device Context?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Create A GL Device Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

if (!(PixelFormat=ChoosePixelFormat(hDC,&pfd))) // Did Windows Find A Matching Pixel Format?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Find A Suitable PixelFormat.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

if(!SetPixelFormat(hDC,PixelFormat,&pfd)) // Are We Able To Set The Pixel Format?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Set The PixelFormat.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

if (!(hRC=wglCreateContext(hDC))) // Are We Able To Get A Rendering Context?
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Can't Create A GL Rendering Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

if(!wglMakeCurrent(hDC,hRC)) // Try To Activate The Rendering Context
{
KillGLWindow(); // Reset The Display
```

```
MessageBox(NULL,"Can't Activate The GL Rendering Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

ShowWindow(hWnd,SW_SHOW); // Show The Window
SetForegroundWindow(hWnd); // Slightly Higher Priority
SetFocus(hWnd); // Sets Keyboard Focus To The Window
ReSizeGLScene(width, height); // Set Up Our Perspective GL Screen

if (!InitGL()) // Initialize Our Newly Created GL Window
{
KillGLWindow(); // Reset The Display
MessageBox(NULL,"Initialization Failed.","ERROR",MB_OK|MB_ICONEXCLAMATION);
return FALSE; // Return FALSE
}

return TRUE; // Success
}
```

WndProc() has not changed, so we will skip over it.

```
LRESULT CALLBACK WndProc( HWND hWnd, // Handle For This Window
UINT uMsg, // Message For This Window
WPARAM wParam, // Additional Message Information
LPARAM lParam) // Additional Message Information
```

Nothing new here. Typical start to WinMain().

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

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

The only real big change in this section of the code is the new window title to let everyone know the tutorial is about reflections using the stencil buffer. Also notice that we pass the resx, resy and resbpp variables to our window creation procedure instead of the usual 640, 480 and 16.

```
// Create Our OpenGL Window
if (!CreateGLWindow("Banu Octavian & NeHe's Stencil & Reflection Tutorial", resx, resy, resbpp,
fullscreen))
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if (active) // Program Active?
{
if (keys[VK_ESCAPE]) // Was Escape Pressed?
{
done=TRUE; // ESC Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
DrawGLScene(); // Draw The Scene
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
```

Instead of checking for key presses in WinMain(), we jump to our keyboard handling routine called ProcessKeyboard(). Notice the ProcessKeyboard() routine is only called if the program is active!

```
ProcessKeyboard(); // Processed Keyboard Presses
}
}
}
}

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

I really hope you've enjoyed this tutorial. I know it could use a little more work. It was one of the more difficult tutorials that I have written. It's easy for me to understand what everything is doing, and what commands I need to use to create cool effects, but when you sit down and actually try to explain things keeping in mind that most people have never even heard of the stencil buffer, it's tough! If you notice anything that could be made clearer or if you find any mistakes in the tutorial please let me know. As always, I want this tutorial to be the best it can possibly be, your feedback is greatly appreciated.

Banu Cosmin (**Choko**)

Jeff Molofee (**NeHe**)

Lesson 27
Shadows

Welcome to a fairly complex tutorial on shadow casting. The effect this demo creates is literally incredible. Shadows that stretch, bend and wrap around other objects and across walls. Everything in the scene can be moved around in 3D space using keys on the keyboard.

This tutorial takes a fairly different approach - It assumes you have a lot of OpenGL knowledge. You should already understand the stencil buffer, and basic OpenGL setup. If you need to brush up, go back and read the earlier tutorials. Functions such as CreateGLWindow and WinMain will NOT be explained in this tutorial. Additionally, some fundamental 3D math is assumed, so keep a good textbook handy! (I used my 1st year maths lecture notes from University - I knew they'd come in handy later on! :)

First we have the definition of INFINITY, which represents how far to extend the shadow volume polygons (this will be explained more later on). If you are using a larger or smaller coordinate system, adjust this value accordingly.

```
// Definition Of "INFINITY" For Calculating The Extension Vector For The Shadow Volume
#define INFINITY 100
```

Next is the definition of the object structures.

The Point3f structure holds a coordinate in 3D space. This can be used for vertices or vectors.

```
// Structure Describing A Vertex In An Object
struct Point3f
{
GLfloat x, y, z;
};
```

The Plane structure holds the 4 values that form the equation of a plane. These planes will represent the faces of the object.

```
// Structure Describing A Plane, In The Format: ax + by + cz + d = 0
struct Plane
{
GLfloat a, b, c, d;
};
```

The Face structure contains all the information necessary about a triangle to cast a shadow.

- The indices specified are from the object's array of vertices.
- The vertex normals are used to calculate the orientation of the face in 3D space, so you can determine which are facing the light source when casting the shadows.
- The plane equation describes the plane that this triangle lies in, in 3D space.
- The neighbour indices are indices into the array of faces in the object. This allows you to specify which face joins this face at each edge of the triangle.
- The visible parameter is used to specify whether the face is "visible" to the light source which is casting the shadows.

```
// Structure Describing An Object's Face
struct Face
{
```

```
int vertexIndices[3]; // Index Of Each Vertex Within An Object That Makes Up The Triangle Of
This Face
Point3f normals[3]; // Normals To Each Vertex
Plane planeEquation; // Equation Of A Plane That Contains This Triangle
int neighbourIndices[3]; // Index Of Each Face That Neighbours This One Within The Object
bool visible; // Is The Face Visible By The Light?
};
```

Finally, the ShadowedObject structure contains all the vertices and faces in the object. The memory for each of the arrays is dynamically created when it is loaded.

```
struct ShadowedObject
{
int nVertices;
Point3f *pVertices; // Will Be Dynamically Allocated

int nFaces;
Face *pFaces; // Will Be Dynamically Allocated
};
```

The readObject function is fairly self explanatory. It will fill in the given object structure with the values read from the file, allocating memory for the vertices and faces. It also initializes the neighbours to -1, which means there isn't one (yet). They will be calculated later.

```
bool readObject( const char *filename, ShadowedObject& object )
{
FILE *pInputFile;
int i;

pInputFile = fopen( filename, "r" );
if ( pInputFile == NULL )
{
cerr << "Unable to open the object file: " << filename << endl;
return false;
}

// Read Vertices
fscanf( pInputFile, "%d", &object.nVertices );
object.pVertices = new Point3f[object.nVertices];
for ( i = 0; i < object.nVertices; i++ )
{
fscanf( pInputFile, "%f", &object.pVertices[i].x );
fscanf( pInputFile, "%f", &object.pVertices[i].y );
fscanf( pInputFile, "%f", &object.pVertices[i].z );
}

// Read Faces
fscanf( pInputFile, "%d", &object.nFaces );
object.pFaces = new Face[object.nFaces];
for ( i = 0; i < object.nFaces; i++ )
{
int j;
Face *pFace = &object.pFaces[i];

for ( j = 0; j < 3; j++ )
pFace->neighbourIndices[j] = -1; // No Neigbours Set Up Yet

for ( j = 0; j < 3; j++ )
{
fscanf( pInputFile, "%d", &pFace->vertexIndices[j] );
pFace->vertexIndices[j]--; // Files Specify Them With A 1 Array Base, But We Use A 0 Array Base
}

for ( j = 0; j < 3; j++ )
{
fscanf( pInputFile, "%f", &pFace->normals[j].x );
fscanf( pInputFile, "%f", &pFace->normals[j].y );
fscanf( pInputFile, "%f", &pFace->normals[j].z );
}
}
return true;
}
```

Likewise, killObject is self-explanatory - just delete all those dynamically allocated arrays inside the object when you are done with them. Note that a line was added to KillGLWindow to call this function for the object in question.

```
void killObject( ShadowedObject& object )
{
delete[] object.pFaces;
object.pFaces = NULL;
object.nFaces = 0;

delete[] object.pVertices;
object.pVertices = NULL;
object.nVertices = 0;
}
```

Now, with setConnectivity it starts to get interesting. This function is used to find out what neighbours there are to each face of the object given. Here's some pseudo code:

```
for each face (A) in the object
for each edge in A
if we don't know this edges neighbour yet
for each face (B) in the object (except A)
for each edge in B
if A's edge is the same as B's edge, then they are neighbouring each other on that edge
set the neighbour property for each face A and B, then move onto next edge in A
```

The last two lines are accomplished with the following code. By finding the two vertices that mark the ends of an edge and comparing them, you can discover if it is the same edge. The part (edgeA+1)%3 gets a vertex next to the one you are considering. Then you check if the vertices match (the order may be different, hence the second case of the if statement).

```
int vertA1 = pFaceA->vertexIndices[edgeA];
int vertA2 = pFaceA->vertexIndices[( edgeA+1 )%3];

int vertB1 = pFaceB->vertexIndices[edgeB];
int vertB2 = pFaceB->vertexIndices[( edgeB+1 )%3];

// Check If They Are Neighbours - IE, The Edges Are The Same
if (( vertA1 == vertB1 && vertA2 == vertB2 ) || ( vertA1 == vertB2 && vertA2 == vertB1 ))
{
pFaceA->neighbourIndices[edgeA] = faceB;
pFaceB->neighbourIndices[edgeB] = faceA;
edgeFound = true;
break;
}
```

Luckily, another easy function while you take a breath. drawObject renders each face one by one.

```
// Draw An Object - Simply Draw Each Triangular Face.
void drawObject( const ShadowedObject& object )
{
glBegin( GL_TRIANGLES );
for ( int i = 0; i < object.nFaces; i++ )
{
const Face& face = object.pFaces[i];

for ( int j = 0; j < 3; j++ )
{
const Point3f& vertex = object.pVertices[face.vertexIndices[j]];

glNormal3f( face.normals[j].x, face.normals[j].y, face.normals[j].z );
glVertex3f( vertex.x, vertex.y, vertex.z );
}
}
glEnd();
}
```

Calculating the equation of a plane looks ugly, but it is just a simple mathematical formula that you grab from a textbook when you need it.

```
void calculatePlane( const ShadowedObject& object, Face& face )
{
// Get Shortened Names For The Vertices Of The Face
const Point3f& v1 = object.pVertices[face.vertexIndices[0]];
const Point3f& v2 = object.pVertices[face.vertexIndices[1]];
const Point3f& v3 = object.pVertices[face.vertexIndices[2]];

face.planeEquation.a = v1.y*(v2.z-v3.z) + v2.y*(v3.z-v1.z) + v3.y*(v1.z-v2.z);
face.planeEquation.b = v1.z*(v2.x-v3.x) + v2.z*(v3.x-v1.x) + v3.z*(v1.x-v2.x);
face.planeEquation.c = v1.x*(v2.y-v3.y) + v2.x*(v3.y-v1.y) + v3.x*(v1.y-v2.y);
face.planeEquation.d = -( v1.x*( v2.y*v3.z - v3.y*v2.z ) +
v2.x*(v3.y*v1.z - v1.y*v3.z) +
v3.x*(v1.y*v2.z - v2.y*v1.z) );
}
```

Have you caught your breath yet? Good, because you are about to learn how to cast a shadow! The castShadow function does all of the GL specifics, and passes it on to doShadowPass to render the shadow in two passes.

First up, we determine which surfaces are facing the light. We do this by seeing which side of the plane the light is on. This is done by substituting the light's position into the equation for the plane. If this is larger than 0, then it is in the same direction as the normal to the plane and visible by the light. If not, then it is not visible by the light. (Again, refer to a good Math textbook for a better explanation of geometry in 3D).

```
void castShadow( ShadowedObject& object, GLfloat *lightPosition )
{
// Determine Which Faces Are Visible By The Light.
for ( int i = 0; i < object.nFaces; i++ )
{
```

```
const Plane& plane = object.pFaces[i].planeEquation;

GLfloat side = plane.a*lightPosition[0]+
plane.b*lightPosition[1]+
plane.c*lightPosition[2]+
plane.d;

if ( side > 0 )
object.pFaces[i].visible = true;
else
object.pFaces[i].visible = false;
}
```

The next section sets up the necessary OpenGL states for rendering the shadows.

First, we push all the attributes onto the stack that will be modified. This makes changing them back a lot easier.

Lighting is disabled because we will not be rendering to the color (output) buffer, just the stencil buffer. For the same reason, the color mask turns off all color components (so drawing a polygon won't get through to the output buffer).

Although depth testing is still used, we don't want the shadows to appear as solid objects in the depth buffer, so the depth mask prevents this from happening.

The stencil buffer is turned on as that is what is going to be used to draw the shadows into.

```
glPushAttrib( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_ENABLE_BIT | GL_POLYGON_BIT |
GL_STENCIL_BUFFER_BIT );
glDisable( GL_LIGHTING ); // Turn Off Lighting
glDepthMask( GL_FALSE ); // Turn Off Writing To The Depth-Buffer
glDepthFunc( GL_LEQUAL );
glEnable( GL_STENCIL_TEST ); // Turn On Stencil Buffer Testing
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE ); // Don't Draw Into The Colour Buffer
glStencilFunc( GL_ALWAYS, 1, 0xFFFFFFFFL );
```

Ok, now the shadows are actually rendered. We'll come back to that in a moment when we look at the doShadowPass function. They are rendered in two passes as you can see, one incrementing the stencil buffer with the front faces (casting the shadow), the second decrementing the stencil buffer with the backfaces ("turning off" the shadow between the object and any other surfaces).

```
// First Pass. Increase Stencil Value In The Shadow
glFrontFace( GL_CCW );
glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );
doShadowPass( object, lightPosition );
// Second Pass. Decrease Stencil Value In The Shadow
glFrontFace( GL_CW );
glStencilOp( GL_KEEP, GL_KEEP, GL_DECR );
doShadowPass( object, lightPosition );
```

To understand how the second pass works, my best advise is to comment it out and run the tutorial again. To save you the trouble, I have done it here:

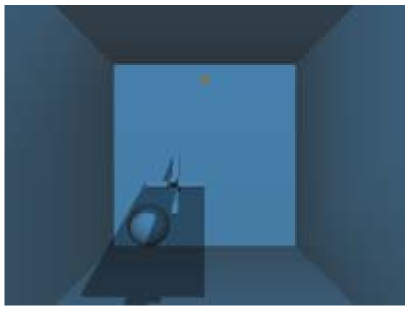


Figure 1: First Pass

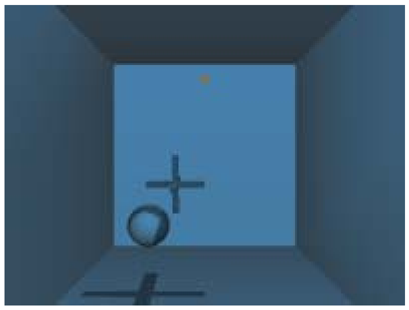


Figure 2: Second Pass

The final section of this function draws one blended rectangle over the whole screen, to cast a shadow. The darker you make this rectangle, the darker the shadows will be. So to change the properties of the shadow, change the glColor4f statement. Higher alpha will make it more black. Or you can make it red, green, purple, ...!

```
glFrontFace( GL_CCW );
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE ); // Enable Rendering To Colour Buffer For All
Components

// Draw A Shadowing Rectangle Covering The Entire Screen
glColor4f( 0.0f, 0.0f, 0.0f, 0.4f );
glEnable( GL_BLEND );
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
```

```
glStencilFunc( GL_NOTEQUAL, 0, 0xFFFFFFFFL );
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
glPushMatrix();
glLoadIdentity();
glBegin( GL_TRIANGLE_STRIP );
glVertex3f(-0.1f, 0.1f,-0.10f);
glVertex3f(-0.1f,-0.1f,-0.10f);
glVertex3f( 0.1f, 0.1f,-0.10f);
glVertex3f( 0.1f,-0.1f,-0.10f);
glEnd();
glPopMatrix();
glPopAttrib();
}
```

Ok, the next part draws the shadowed quads. How does that work? What happens is that you go through every face, and if it is visible, then you check all of its edges. If at the edge, there is no neighbouring face, or the neighbouring face is not visible, the edge casts a shadow. If you think about the two cases clearly, then you'll see this is true. By drawing a quadrilateral (as two triangles) comprising of the points of the edge, and the edge projected backwards through the scene you get the shadow cast by it.

The brute force approach used here just draws to "infinity", and the shadow polygon is clipped against all the polygons it encounters. This causes piercing, which will stress the video hardware. For a high-performance modification to this algorithm, you should clip the polygon to the objects behind it. This is much trickier and has problems of its own, but if that's what you want to do, you should refer to this Gamasutra article.

The code to do all of that is not as tricky as it sounds. To start with, here is a snippet that loops through the objects. By the end of it, we have an edge, *j*, and its neighbouring face, specified by *neighbourIndex*.

```
void doShadowPass( ShadowedObject& object, GLfloat *lightPosition )
{
for ( int i = 0; i < object.nFaces; i++ )
{
const Face& face = object.pFaces[i];

if ( face.visible )
{
// Go Through Each Edge
for ( int j = 0; j < 3; j++ )
{
int neighbourIndex = face.neighbourIndices[j];
```

Next, check if there is a visible neighbouring face to this object. If not, then this edge casts a shadow.

```
// If There Is No Neighbour, Or Its Neighbouring Face Is Not Visible, Then This Edge Casts A
Shadow
if ( neighbourIndex == -1 || object.pFaces[neighbourIndex].visible == false )
{
```

The next segment of code will retrieve the two vertices from the current edge, *v1* and *v2*. Then, it calculates *v3* and *v4*, which are projected along the vector between the light source and the first edge. They are scaled to INFINITY, which was set to a very large value.

```
// Get The Points On The Edge
const Point3f& v1 = object.pVertices[face.vertexIndices[j]];
const Point3f& v2 = object.pVertices[face.vertexIndices[( j+1 )%3]];

// Calculate The Two Vertices In Distance
Point3f v3, v4;

v3.x = ( v1.x-lightPosition[0] )*INFINITY;
v3.y = ( v1.y-lightPosition[1] )*INFINITY;
v3.z = ( v1.z-lightPosition[2] )*INFINITY;

v4.x = ( v2.x-lightPosition[0] )*INFINITY;
v4.y = ( v2.y-lightPosition[1] )*INFINITY;
v4.z = ( v2.z-lightPosition[2] )*INFINITY;
```

I think you'll understand the next section, it justs draws the quadrilateral defined by those four points:

```
// Draw The Quadrilateral (As A Triangle Strip)
glBegin( GL_TRIANGLE_STRIP );
glVertex3f( v1.x, v1.y, v1.z );
glVertex3f( v1.x+v3.x, v1.y+v3.y, v1.z+v3.z );
glVertex3f( v2.x, v2.y, v2.z );
glVertex3f( v2.x+v4.x, v2.y+v4.y, v2.z+v4.z );
glEnd();
}
}
}
}
}
```

With that, the shadow casting section is completed. But we are not finished yet! What about drawGLScene? Lets start with the simple bits: clearing the buffers, positioning the light source, and drawing a sphere:

```
bool drawGLScene()
{
GLmatrix16f Minv;
GLvector4f wlp, lp;

// Clear Color Buffer, Depth Buffer, Stencil Buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

glLoadIdentity(); // Reset Modelview Matrix
glTranslatef(0.0f, 0.0f, -20.0f); // Zoom Into Screen 20 Units
glLightfv(GL_LIGHT1, GL_POSITION, LightPos); // Position Light1
glTranslatef(SpherePos[0], SpherePos[1], SpherePos[2]); // Position The Sphere
gluSphere(q, 1.5f, 32, 16); // Draw A Sphere
```

Next, we have to calculate the light's position relative to the local coordinate system of the object. The comments explain each step in detail. *Minv* stores the object's transformation matrix, however it is done in reverse, and with negative arguments, so it is actually the inverse of the transformation matrix. Then *lp* is created as a copy of the light's position, and multiplied by the matrix. Thus, *lp* is the light's position in the object's coordinate system.

```
glLoadIdentity(); // Reset Matrix
glRotatef(-yrot, 0.0f, 1.0f, 0.0f); // Rotate By -yrot On Y Axis
glRotatef(-xrot, 1.0f, 0.0f, 0.0f); // Rotate By -xrot On X Axis
glTranslatef(-ObjPos[0], -ObjPos[1], -ObjPos[2]); // Move Negative On All Axis Based On ObjPos[]
Values (X, Y, Z)
glGetFloatv(GL_MODELVIEW_MATRIX,Minv); // Retrieve ModelView Matrix (Stores In Minv)
lp[0] = LightPos[0]; // Store Light Position X In lp[0]
lp[1] = LightPos[1]; // Store Light Position Y In lp[1]
lp[2] = LightPos[2]; // Store Light Position Z In lp[2]
lp[3] = LightPos[3]; // Store Light Direction In lp[3]
VMatMult(Minv, lp); // We Store Rotated Light Vector In 'lp' Array
```

Now, palm off some of the work to draw the room, and the object. Calling castShadow draws the shadow of the object.

```
glLoadIdentity(); // Reset Modelview Matrix
glTranslatef(0.0f, 0.0f, -20.0f); // Zoom Into The Screen 20 Units
DrawGLRoom(); // Draw The Room
glTranslatef(ObjPos[0], ObjPos[1], ObjPos[2]); // Position The Object
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Spin It On The X Axis By xrot
glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Spin It On The Y Axis By yrot
drawObject(obj); // Procedure For Drawing The Loaded Object
castShadow(obj, lp); // Procedure For Casting The Shadow Based On The Silhouette
```

The following few lines draw a little orange circle where the light is:

```
glColor4f(0.7f, 0.4f, 0.0f, 1.0f); // Set Color To An Orange
glDisable(GL_LIGHTING); // Disable Lighting
glDepthMask(GL_FALSE); // Disable Depth Mask
glTranslatef(lp[0], lp[1], lp[2]); // Translate To Light's Position
// Notice We're Still In Local Coordinate System
gluSphere(q, 0.2f, 16, 8); // Draw A Little Yellow Sphere (Represents Light)
glEnable(GL_LIGHTING); // Enable Lighting
glDepthMask(GL_TRUE); // Enable Depth Mask
```

The last part updates the object's position and returns.

```
xrot += xspeed; // Increase xrot By xspeed
yrot += yspeed; // Increase yrot By yspeed

glFlush(); // Flush The OpenGL Pipeline
return TRUE; // Everything Went OK
}
```

We did specify a DrawGLRoom function, and here it is - a bunch of rectangles to cast shadows against:

```
void DrawGLRoom() // Draw The Room (Box)
{
glBegin(GL_QUADS); // Begin Drawing Quads
// Floor
glNormal3f(0.0f, 1.0f, 0.0f); // Normal Pointing Up
glVertex3f(-10.0f,-10.0f,-20.0f); // Back Left
glVertex3f(-10.0f,-10.0f, 20.0f); // Front Left
glVertex3f( 10.0f,-10.0f, 20.0f); // Front Right
glVertex3f( 10.0f,-10.0f,-20.0f); // Back Right
// Ceiling
glNormal3f(0.0f,-1.0f, 0.0f); // Normal Point Down
glVertex3f(-10.0f, 10.0f, 20.0f); // Front Left
glVertex3f(-10.0f, 10.0f,-20.0f); // Back Left
glVertex3f( 10.0f, 10.0f,-20.0f); // Back Right
glVertex3f( 10.0f, 10.0f, 20.0f); // Front Right
// Front Wall
glNormal3f(0.0f, 0.0f, 1.0f); // Normal Pointing Away From Viewer
glVertex3f(-10.0f, 10.0f,-20.0f); // Top Left
glVertex3f(-10.0f,-10.0f,-20.0f); // Bottom Left
glVertex3f( 10.0f,-10.0f,-20.0f); // Bottom Right
glVertex3f( 10.0f, 10.0f,-20.0f); // Top Right
// Back Wall
glNormal3f(0.0f, 0.0f,-1.0f); // Normal Pointing Towards Viewer
glVertex3f( 10.0f, 10.0f, 20.0f); // Top Right
glVertex3f( 10.0f,-10.0f, 20.0f); // Bottom Right
glVertex3f(-10.0f,-10.0f, 20.0f); // Bottom Left
glVertex3f(-10.0f, 10.0f, 20.0f); // Top Left
// Left Wall
glNormal3f(1.0f, 0.0f, 0.0f); // Normal Pointing Right
glVertex3f(-10.0f, 10.0f, 20.0f); // Top Front
glVertex3f(-10.0f,-10.0f, 20.0f); // Bottom Front
glVertex3f(-10.0f,-10.0f,-20.0f); // Bottom Back
glVertex3f(-10.0f, 10.0f,-20.0f); // Top Back
// Right Wall
glNormal3f(-1.0f, 0.0f, 0.0f); // Normal Pointing Left
glVertex3f( 10.0f, 10.0f,-20.0f); // Top Back
glVertex3f( 10.0f,-10.0f,-20.0f); // Bottom Back
glVertex3f( 10.0f,-10.0f, 20.0f); // Bottom Front
glVertex3f( 10.0f, 10.0f, 20.0f); // Top Front
glEnd(); // Done Drawing Quads
}
```

And before I forget, here is the VMatMult function which multiplies a vector by a matrix (get that Math textbook out again!):

```
void VMatMult(GLmatrix16f M, GLvector4f v)
{
GLfloat res[4]; // Hold Calculated Results
res[0]=M[ 0]*v[0]+M[ 4]*v[1]+M[ 8]*v[2]+M[12]*v[3];
res[1]=M[ 1]*v[0]+M[ 5]*v[1]+M[ 9]*v[2]+M[13]*v[3];
res[2]=M[ 2]*v[0]+M[ 6]*v[1]+M[10]*v[2]+M[14]*v[3];
res[3]=M[ 3]*v[0]+M[ 7]*v[1]+M[11]*v[2]+M[15]*v[3];
v[0]=res[0]; // Results Are Stored Back In v[]
v[1]=res[1];
v[2]=res[2];
v[3]=res[3]; // Homogenous Coordinate
}
```

The function to load the object is simple, just calling readObject, and then setting up the connectivity and the plane equations for each face.

```
int InitGLObjects() // Initialize Objects
{
if (!readObject("Data/Object2.txt", obj)) // Read Object2 Into obj
{
return FALSE; // If Failed Return False
}

setConnectivity(obj); // Set Face To Face Connectivity

for ( int i=0;i < obj.nFaces;i++) // Loop Through All Object Faces
calculatePlane(obj, obj.pFaces[i]); // Compute Plane Equations For All Faces

return TRUE; // Return True
}
```

Finally, KillGLObjects is a convenience function so that if you add more objects, you can add them in a central place.

```
void KillGLObjects()
{
killObject( obj );
}
```

All of the other functions don't require any further explanantion. I have left out the standard NeHe tutorial code, as well as all of the variable definitions and the keyboard processing function. The commenting alone explains these sufficiently.

Some things to note about the tutorial:

- The sphere doesn't stop shadows being projected on the wall. In reality, the sphere should also be casting a shadow, so seeing the one on the wall won't matter, it's hidden. It's just there to see what happens on curved surfaces :)
- If you are noticing extremely slow frame rates, try switching to fullscreen mode, or setting your desktop colour depth to 32bpp.
- Arseny L. writes: If you are having problems with a TNT2 in Windowed mode, make sure your desktop color depth is not set to 16bit. In 16bit color mode, the stencil buffer is emulated, resulting in sluggish performance. There are no problems in 32bit mode (I have a TNT2 Ultra and I checked it).

I've got to admit this was a lengthy task to write out this tutorial. It gives you full appreciation for the work that Jeff puts in! I hope you enjoy it, and give a huge thanks to Banu who wrote the original code! IF there is anything that needs further explaining in here, you are welcome to contact me (Brett), at brettporter@yahoo.com.

Banu Cosmin (**Choko**) & **Brett Porter**

Jeff Molofee (**NeHe**)

Lesson 28
Bezier Patches / Fullscreen Fix

Bezier Patches

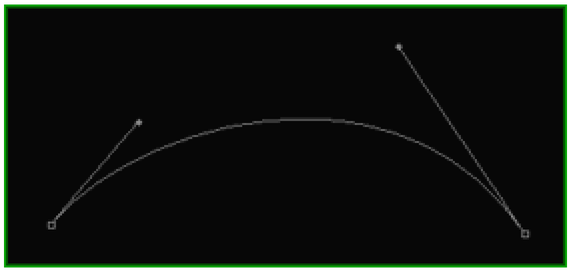
Written by: David Nikdel (ogapo@ithink.net)

This tutorial is intended to introduce you to Bezier Surfaces in the hopes that someone more artistic than myself will do something really cool with them and show all of us. This is not intended as a complete Bezier patch library, but more as proof of concept code to get you familiar with how these curved surfaces actually work. Also, as this is a very informal piece, I may have occasional lapses in correct terminology in favor of comprehensability; I hope this sits well with everyone. Finally, to those of you already familiar with Beziers who are just reading this to see if I screw up, shame on you ;-), but if you find anything wrong by all means let me or NeHe know, after all no one's perfect, eh? Oh, and one more thing, none of this code is optimised beyond my normal programming technique, this is by design. I want everyone to be able to see exactly what is going on. Well, I guess that's enough of an intro. On with the show!

The Math - ::evil music:: (warning, kinda long section)

Ok, it will be very hard to understand Beziers without at least a basic understanding of the math behind it, however, if you just don't feel like reading this section or already know the math, you can skip it. First I will start out by describing the Bezier curve itself then move on to how to create a Bezier Patch.

Odds are, if you've ever used a graphics program you are already familiar with Bezier curves, perhaps not by that name though. They are the primary method of drawing curved lines and are commonly represented as a series of points each with 2 points representing the tangent at that point from the left and right. Here's what one looks like:



This is the most basic Bezier curve possible (longer ones are made by attaching many of these together (many times without the user realizing it)). This curve is actually defined by only 4 points, those would be the 2 ending control points and the 2 middle control points. To the computer, all the points are the same, but to aid in design we often connect the first and the last two, respectively, because those lines will always be tangent to the endpoint. The curve is a parametric curve and is drawn by finding any number of points evenly spaced along the curve and connecting them with straight lines. In this way you can control the resolution of the patch (and the amount of computation). The most common way to use this is to tesselate it less at a farther distance and more at a closer distance so that, to the viewer, it always appears to be a perfectly curved surface with the lowest possible speed hit.

Bezier curves are based on a basis function from which more complicated versions are derived. Here's the function:

t + (1 - t) = 1

Sounds simple enough huh? Well it really is, this is the Bezier most basic Bezier curve, a 1st degree curve. As you may have guessed from the terminology, the Bezier curves are polynomials, and as we remember from algebra, a 1st degree polynomial is just a straight line; not very interesting. Well, since the basis function is true for all numbers t, we can square, cube, whatever, each side and it will still be true right? Well, lets try cubing it.

(t + (1-t))^3 = 1^3

t^3 + 3*t^2*(1-t) + 3*t*(1-t)^2 + (1-t)^3 = 1

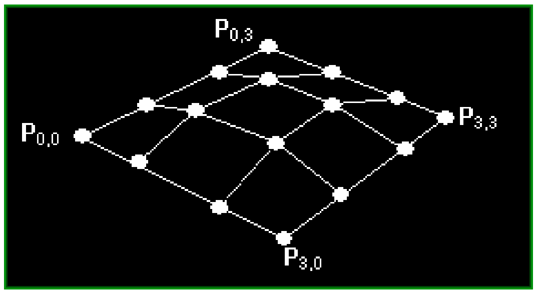
This is the equation we use to calculate the most common Bezier, the 3rd degree Bezier curve. This is most common for two reasons, a) it's the lowest degree polynomial that need not necesarily lie in a plane (there are 4 control points) and b) the tangent lines on the sides are not dependant on one another (with a 2nd degree there would be only 3 control points). So do you see the Bezier curve yet? Hehe, me neither, that's because I still need to add one thing.

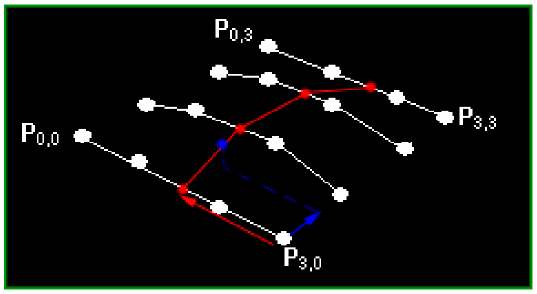
Ok, since the entire left side is equal to 1, it's safe to assume that if you add all the components they should still equal one. Does this sound like it could be used to descide how much of each control point to use in calculating a point on the curve? (hint: just say yes ;-)) Well you're right! When we want to calculate the value of a point some percent along the curve we simply multiply each part by a control point (as a vector) and find the sum. Generally, we'll work with 0 <= t <= 1, but it's not technically necesary. Confused yet? Here's the function:

P1*t^3 + P2*3*t^2*(1-t) + P3*3*t*(1-t)^2 + P4*(1-t)^3 = P_{new}

Because polynomials are always continuous, this makes for a good way to morp between the 4 points. The only points it actually reaches though are P1 and P4, when t = 1 and 0 respectively.

Now, that's all well and good, but how can I use these in 3D you ask? Well it's actually quite simple, in order to form a Bezier patch, you need 16 control points (4*4), and 2 variables t and v. What you do from there is calculate a point at v along 4 of the parallel curves then use those 4 points to make a new curve and calculate t along that curve. By calculating enough of these points, we can draw triangle strips to connect them, thus drawing the Bezier patch.





Well, I suppose that's enough math for now, on to the code!

```
#include <windows.h> // Header File For Windows
#include <math.h> // Header File For Math Library Routines
#include <stdio.h> // Header File For Standard I/O Routines
#include <stdlib.h> // Header File For Standard Library
#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

typedef struct point_3d { // Structure For A 3-Dimensional Point ( NEW )
double x, y, z;
} POINT_3D;

typedef struct bpatch { // Structure For A 3rd Degree Bezier Patch ( NEW )
POINT_3D anchors[4][4]; // 4x4 Grid Of Anchor Points
GLuint dlBPatch; // Display List For Bezier Patch
GLuint texture; // Texture For The Patch
} BEZIER_PATCH;

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

DEVMODE DMsaved; // Saves The Previous Screen Settings ( NEW )

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

GLfloat rotz = 0.0f; // Rotation About The Z Axis
BEZIER_PATCH mybezier; // The Bezier Patch We're Going To Use ( NEW )
BOOL showCPoints=TRUE; // Toggles Displaying The Control Point Grid ( NEW )
```

```
int divs = 7; // Number Of Intrapolations (Controls Poly Resolution) ( NEW )

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

The following are just a few quick functions for some simple vector math. If you're a fan of C++ you might consider using a point class (just make sure it's 3d).

```
// Adds 2 Points. Don't Just Use '+' ;)
POINT_3D pointAdd(POINT_3D p, POINT_3D q) {
p.x += q.x; p.y += q.y; p.z += q.z;
return p;
}

// Multiplies A Point And A Constant. Don't Just Use '*'
POINT_3D pointTimes(double c, POINT_3D p) {
p.x *= c; p.y *= c; p.z *= c;
return p;
}

// Function For Quick Point Creation
POINT_3D makePoint(double a, double b, double c) {
POINT_3D p;
p.x = a; p.y = b; p.z = c;
return p;
}
```

This is basically just the 3rd degree basis function written in C, it takes a variable u and an array of 4 points and computes a point on the curve. By stepping u in equal increments between 0 and 1, we'll get a nice approximation of the curve.

```
// Calculates 3rd Degree Polynomial Based On Array Of 4 Points
// And A Single Variable (u) Which Is Generally Between 0 And 1
POINT_3D Bernstein(float u, POINT_3D *p) {
POINT_3D a, b, c, d, r;

a = pointTimes(pow(u,3), p[0]);
b = pointTimes(3*pow(u,2)*(1-u), p[1]);
c = pointTimes(3*u*pow((1-u),2), p[2]);
d = pointTimes(pow((1-u),3), p[3]);

r = pointAdd(pointAdd(a, b), pointAdd(c, d));

return r;
}
```

This function does the lion's share of the work by generating all the triangle strips and storing them in a display list. We do this so that we don't have to recalculate the patch each frame, only when it changes. By the way, a cool effect you might want to try might be to use the morphing tutorial to morph the patch's control points. This would yeild a very cool smooth, organic, morphing effect for relatively little overhead (you only morph 16 points, but you have to recalculate). The "last" array is used to keep the previous line of points (since a triangle strip needs both rows). Also, texture coordinates are calculated by using the u and v values as the percentages (planar mapping).

One thing we don't do is calculate the normals for lighting. When it comes to this, you basically have two options. The first is to find the center of each triangle, then use a bit of calculus and calculate the tangent on both the x and y axes, then do the cross product to get a vector perpendicular to both, THEN normalize the vector and use that as the normal. OR (yes, there is a faster way) you can cheat and just use the normal of the triangle (calculated your favorite way) to get a pretty good approximation. I prefer the latter; the speed hit, in my opinion, isn't worth the extra little bit of realism.

```
// Generates A Display List Based On The Data In The Patch
// And The Number Of Divisions
GLuint genBezier(BEZIER_PATCH patch, int divs) {
int u = 0, v;
float py, px, pyold;
GLuint drawlist = glGenLists(1); // Make The Display List
POINT_3D temp[4];
POINT_3D *last = (POINT_3D*)malloc(sizeof(POINT_3D)*(divs+1));
// Array Of Points To Mark The First Line Of Polys

if (patch.dlBPatch != NULL) // Get Rid Of Any Old Display Lists
glDeleteLists(patch.dlBPatch, 1);

temp[0] = patch.anchors[0][3]; // The First Derived Curve (Along X-Axis)
temp[1] = patch.anchors[1][3];
temp[2] = patch.anchors[2][3];
temp[3] = patch.anchors[3][3];

for (v=0;v<=divs;v++) { // Create The First Line Of Points
px = ((float)v)/((float)divs); // Percent Along Y-Axis
// Use The 4 Points From The Derived Curve To Calculate The Points Along That Curve
last[v] = Bernstein(px, temp);
}

glNewList(drawlist, GL_COMPILE); // Start A New Display List
glBindTexture(GL_TEXTURE_2D, patch.texture); // Bind The Texture
for (u=1;u<=divs;u++) {
py = ((float)u)/((float)divs); // Percent Along Y-Axis
pyold = ((float)u-1.0f)/((float)divs); // Percent Along Old Y Axis

temp[0] = Bernstein(py, patch.anchors[0]); // Calculate New Bezier Points
temp[1] = Bernstein(py, patch.anchors[1]);
temp[2] = Bernstein(py, patch.anchors[2]);
temp[3] = Bernstein(py, patch.anchors[3]);

glBegin(GL_TRIANGLE_STRIP); // Begin A New Triangle Strip

for (v=0;v<=divs;v++) {
px = ((float)v)/((float)divs); // Percent Along The X-Axis

glTexCoord2f(pyold, px); // Apply The Old Texture Coords
glVertex3d(last[v].x, last[v].y, last[v].z); // Old Point

last[v] = Bernstein(px, temp); // Generate New Point
glTexCoord2f(py, px); // Apply The New Texture Coords
glVertex3d(last[v].x, last[v].y, last[v].z); // New Point
}

glEnd(); // END The Triangle Strip
}

glEndList(); // END The List

free(last); // Free The Old Vertices Array
return drawlist; // Return The Display List
}
```

Here we're just loading the matrix with some values I've picked that I think look cool. Feel free to screw around with these and see what it looks like. :-)

```
void initBezier(void) {
mybezier.anchors[0][0] = makePoint(-0.75, -0.75, -0.50); // Set The Bezier Vertices
mybezier.anchors[0][1] = makePoint(-0.25, -0.75, 0.00);
mybezier.anchors[0][2] = makePoint( 0.25, -0.75, 0.00);
mybezier.anchors[0][3] = makePoint( 0.75, -0.75, -0.50);
mybezier.anchors[1][0] = makePoint(-0.75, -0.25, -0.75);
mybezier.anchors[1][1] = makePoint(-0.25, -0.25, 0.50);
mybezier.anchors[1][2] = makePoint( 0.25, -0.25, 0.50);
mybezier.anchors[1][3] = makePoint( 0.75, -0.25, -0.75);
mybezier.anchors[2][0] = makePoint(-0.75, 0.25, 0.00);
mybezier.anchors[2][1] = makePoint(-0.25, 0.25, -0.50);
mybezier.anchors[2][2] = makePoint( 0.25, 0.25, -0.50);
mybezier.anchors[2][3] = makePoint( 0.75, 0.25, 0.00);
mybezier.anchors[3][0] = makePoint(-0.75, 0.75, -0.50);
mybezier.anchors[3][1] = makePoint(-0.25, 0.75, -1.00);
mybezier.anchors[3][2] = makePoint( 0.25, 0.75, -1.00);
mybezier.anchors[3][3] = makePoint( 0.75, 0.75, -0.50);
mybezier.dlBPatch = NULL; // Go Ahead And Initialize This To NULL
}
```

This is basically just an optimised routine to load a single bitmap. It can easily be used to load an array of em just by putting it in a simple loop.

```
// Load Bitmaps And Convert To Textures

BOOL LoadGLTexture(GLuint *texPntr, char* name)
{
BOOL success = FALSE;
AUX_RGBImageRec *TextureImage = NULL;

glGenTextures(1, texPntr); // Generate 1 Texture

FILE* test=NULL;
TextureImage = NULL;

test = fopen(name, "r"); // Test To See If The File Exists
if (test != NULL) { // If It Does
fclose(test); // Close The File
TextureImage = auxDIBImageLoad(name); // And Load The Texture
}

if (TextureImage != NULL) { // If It Loaded
success = TRUE;

// Typical Texture Generation Using Data From The Bitmap
glBindTexture(GL_TEXTURE_2D, *texPntr);
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage->sizeX, TextureImage->sizeY, 0, GL_RGB,
GL_UNSIGNED_BYTE, TextureImage->data);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
}
if (TextureImage->data)
free(TextureImage->data);

return success;
}
```

Just adding the patch initialization here. You would do this whenever you create a patch. Again, this might be a cool place to use C++ (bezier class?).

```
int InitGL(GLvoid) // All Setup For OpenGL Goes Here
{
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glShadeModel(GL_SMOOTH); // Enable Smooth Shading
glClearColor(0.05f, 0.05f, 0.05f, 0.5f); // Black Background
glClearDepth(1.0f); // Depth Buffer Setup
glEnable(GL_DEPTH_TEST); // Enables Depth Testing
glDepthFunc(GL_LEQUAL); // The Type Of Depth Testing To Do
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Really Nice Perspective Calculations

initBezier(); // Initialize the Bezier's Control Grid ( NEW )
LoadGLTexture(&(mybezier.texture), "./Data/NeHe.bmp"); // Load The Texture ( NEW )
mybezier.dlBPatch = genBezier(mybezier, divs); // Generate The Patch ( NEW )

return TRUE; // Initialization Went OK
}
```

First call the bezier's display list. Then (if the outlines are on) draw the lines connecting the control points. You can toggle these by pressing SPACE.

```
int DrawGLScene(GLvoid) { // Here's Where We Do All The Drawing
int i, j;
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
glLoadIdentity(); // Reset The Current Modelview Matrix
glTranslatef(0.0f,0.0f,-4.0f); // Move Left 1.5 Units And Into The Screen 6.0
glRotatef(-75.0f,1.0f,0.0f,0.0f);
glRotatef(rotz,0.0f,0.0f,1.0f); // Rotate The Triangle On The Z-Axis

glCallList(mybezier.dlBPatch); // Call The Bezier's Display List
// This Need Only Be Updated When The Patch Changes

if (showCPoints) { // If Drawing The Grid Is Toggled On
glDisable(GL_TEXTURE_2D);
glColor3f(1.0f,0.0f,0.0f);
for(i=0;i<4;i++) { // Draw The Horizontal Lines
glBegin(GL_LINE_STRIP);
for(j=0;j<4;j++)
glVertex3d(mybezier.anchors[i][j].x, mybezier.anchors[i][j].y, mybezier.anchors[i][j].z);
glEnd();
}
for(i=0;i<4;i++) { // Draw The Vertical Lines
glBegin(GL_LINE_STRIP);
for(j=0;j<4;j++)
glVertex3d(mybezier.anchors[j][i].x, mybezier.anchors[j][i].y, mybezier.anchors[j][i].z);
glEnd();
}
glColor3f(1.0f,1.0f,1.0f);
glEnable(GL_TEXTURE_2D);
}

return TRUE; // Keep Going
}
```

This function contains some modified code to make your projects more compatable. It doesn't have anything to do with Bezier curves, but it does fix a problem with switching back the resolution after fullscreen mode with some video cards (including mine, a crappy old ATI Rage PRO, and a few others). I hope, you'll use this from now on so me and others with similar cards can view your cool examples GL code properly. To make these modifications make the changes in KillGLWindow(), make sure and define DMsaved, and make the one line change in CreateGLWindow() (it's marked).

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
{
if (fullscreen) // Are We In Fullscreen Mode?
{
if (!ChangeDisplaySettings(NULL,CDS_TEST)) { // If The Shortcut Doesn't Work ( NEW )
ChangeDisplaySettings(NULL,CDS_RESET); // Do It Anyway (To Get The Values Out Of The Registry) (
NEW )
ChangeDisplaySettings(&DMsaved,CDS_RESET); // Change It To The Saved Settings ( NEW )
} else {
ChangeDisplaySettings(NULL,CDS_RESET); // If It Works, Go Right Ahead ( NEW )
}

ShowCursor(TRUE); // Show Mouse Pointer
}

if (hRC) // Do We Have A Rendering Context?
{
if (!wglMakeCurrent(NULL,NULL)) // Are We Able To Release The DC And RC Contexts?
{
MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
}

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

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

if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?
{
MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hWnd=NULL; // Set hWnd To NULL
}

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

Just added the EnumDisplaySettings() command here to save the old display settings. (part of the old graphics card fix).

```
// This Code Creates Our OpenGL Window. Parameters Are: *
// title - Title To Appear At The Top Of The Window *
// width - Width Of The GL Window Or Fullscreen Mode *
// height - Height Of The GL Window Or Fullscreen Mode *
// bits - Number Of Bits To Use For Color (8/16/24/32) *
// fullscreenflag - Use Fullscreen Mode (TRUE) Or Windowed Mode (FALSE) */

BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
GLuint PixelFormat; // Holds The Results After Searching For A Match
WNDCLASS wc; // Windows Class Structure
DWORD dwExStyle; // Window Extended Style
DWORD dwStyle; // Window Style
RECT WindowRect; // Grabs Rectangle Upper Left / Lower Right Values
WindowRect.left=(long)0; // Set Left Value To 0
WindowRect.right=(long)width; // Set Right Value To Requested Width
WindowRect.top=(long)0; // Set Top Value To 0
WindowRect.bottom=(long)height; // Set Bottom Value To Requested Height

fullscreen=fullscreenflag; // Set The Global Fullscreen Flag

hInstance = GetModuleHandle(NULL); // Grab An Instance For Our Window
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // Redraw On Size, And Own DC For Window
wc.lpfnWndProc = (WNDPROC) WndProc; // WndProc Handles Messages
wc.cbClsExtra = 0; // No Extra Window Data
wc.cbWndExtra = 0; // No Extra Window Data
wc.hInstance = hInstance; // Set The Instance
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Load The Default Icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Load The Arrow Pointer
wc.hbrBackground = NULL; // No Background Required For GL
wc.lpszMenuName = NULL; // We Don't Want A Menu
wc.lpszClassName = "OpenGL"; // Set The Class Name

EnumDisplaySettings(NULL, ENUM_CURRENT_SETTINGS, &DMsaved); // Save The Current Display State (
NEW )

if (fullscreen) // Attempt Fullscreen Mode?
{
DEVMODE dmScreenSettings; // Device Mode
memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's Cleared
dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode Structure
dmScreenSettings.dmPelsWidth = width; // Selected Screen Width
dmScreenSettings.dmPelsHeight = height; // Selected Screen Height
dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pixel
dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

... Code Cut To Save Space (No Further Changes To This Function) ...

return TRUE; // Success
}
```

All I did here was add commands to rotate the patch, raise/lower the resolution, and toggle the control lines.

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}

// Create Our OpenGL Window
if (!CreateGLWindow("NeHe's Solid Object Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit Received?
{
done=TRUE; // ESC or DrawGLScene Signalled A Quit
}
else // Not Time To Quit, Update Screen
{
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
}


if (keys[VK_LEFT]) rotz -= 0.8f; // Rotate Left ( NEW )
if (keys[VK_RIGHT]) rotz += 0.8f; // Rotate Right ( NEW )
if (keys[VK_UP]) { // Resolution Up ( NEW )
divs++;
mybezier.dlBPatch = genBezier(mybezier, divs); // Update The Patch
keys[VK_UP] = FALSE;
}
if (keys[VK_DOWN] && divs > 1) { // Resolution Down ( NEW )
divs--;
mybezier.dlBPatch = genBezier(mybezier, divs); // Update The Patch
keys[VK_DOWN] = FALSE;
}
if (keys[VK_SPACE]) { // SPACE Toggles showCPoints ( NEW )
showCPoints = !showCPoints;
keys[VK_SPACE] = FALSE;
}


if (keys[VK_F1]) // Is F1 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe's Solid Object Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
}
}

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

Well, I hope this tutorial has been enlightening and you all now love Bezier curves as much as I do ;-). If you like this tutorial I may write another one on NURBS curves if anyone's interested. Please e-mail me and let me know what you thought of this tutorial.

About The Author: David Nikdel is currently 18 and a senior at Bartow Senior High School. His current projects include a research paper on curved surfaces in 3D graphics, an OpenGL based game called Blazing Sands and being lazy. His hobbies include programming, football, and paintballing. He will (hopefully) be a freshman at Georgia Tech next year.

David Nikdel

Jeff Molofee (**NeHe**)

Lesson 29
Blitter Function, RAW Texture Loading

This tutorial was originally written by Andreas Löffler. He also wrote all of the original HTML for the tutorial. A few days later Rob Fletcher emailed me an Irix version of lesson 29. In his version he rewrote most of the code. So I ported Rob's Irix / GLUT code to Visual C++ / Win32. I then modified the message loop code, and the fullscreen code. When the program is minimized it should use 0% of the CPU (or close to). When switching to and from fullscreen mode, most of the problems should be gone (screen not restoring properly, messed up display, etc).

Andreas tutorial is now better than ever. Unfortunately, the code has been modifed quite a bit, so all of the HTML has been rewritten by myself. Huge Thanks to Andreas for getting the ball rolling, and working his butt off to make a killer tutorial. Thanks to Rob for the modifications!

Lets begin... We create a device mode structure called DMsaved. We will use this structure to store information about the users default desktop resolution, color depth, etc., before we switch to fullscreen mode. More on this later! Notice we only allocate enough storage space for one texture (texture[1]).

```
#include <windows.h> // Header File For Windows
#include <gl\gl.h> // Header File For The OpenGL32 Library
#include <gl\glu.h> // Header File For The GLu32 Library
#include <stdio.h> // Header File For File Operation Needed

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

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

DEVMODE DMsaved; // Saves The Previous Screen Settings (NEW)

GLfloat xrot; // X Rotation
GLfloat yrot; // Y Rotation
GLfloat zrot; // Z Rotation

GLuint texture[1]; // Storage For 1 Texture
```

Now for the fun stuff. We create a structure called TEXTURE_IMAGE. The structure contains information about our images width, height, and format (bytes per pixel). data is a pointer to unsigned char. Later on data will point to our image data.

```
typedef struct Texture_Image
{
int width; // Width Of Image In Pixels
int height; // Height Of Image In Pixels
int format; // Number Of Bytes Per Pixel
unsigned char *data; // Texture Data
} TEXTURE_IMAGE;
```

We then create a pointer called P_TEXTURE_IMAGE to the TEXTURE_IMAGE data type. The variables t1 and t2 are of type P_TEXTURE_IMAGE where P_TEXTURE_IMAGE is a redefined type of pointer to TEXTURE_IMAGE.

```
typedef TEXTURE_IMAGE *P_TEXTURE_IMAGE; // A Pointer To The Texture Image Data Type

P_TEXTURE_IMAGE t1; // Pointer To The Texture Image Data Type
P_TEXTURE_IMAGE t2; // Pointer To The Texture Image Data Type

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

Below is the code to allocate memory for a texture. When we call this code, we pass it the width, height and bytes per pixel information of the image we plan to load. ti is a pointer to our TEXTURE_IMAGE data type. It's given a NULL value. c is a pointer to unsigned char, it is also set to NULL.

```
// Allocate An Image Structure And Inside Allocate Its Memory Requirements
P_TEXTURE_IMAGE AllocateTextureBuffer( GLint w, GLint h, GLint f)
{
P_TEXTURE_IMAGE ti=NULL; // Pointer To Image Struct
unsigned char *c=NULL; // Pointer To Block Memory For Image
```

Here is where we allocate the memory for our image structure. If everything goes well, ti will point to the allocated memory.

After allocating the memory, and checking to make sure ti is not equal to NULL, we can fill the structure with the image attributes. First we set the width (w), then the height (h) and lastly the format (f). Keep in mind format is bytes per pixel.

```
ti = (P_TEXTURE_IMAGE)malloc(sizeof(TEXTURE_IMAGE)); // One Image Struct Please

if( ti != NULL ) {
ti->width = w; // Set Width
ti->height = h; // Set Height
ti->format = f; // Set Format
```

Now we need to allocate memory for the actual image data. The calculation is easy! We multiply the width of the image (w) by the height of the image (h) then multiply by the format (f - bytes per pixel).

```
c = (unsigned char *)malloc( w * h * f);
```

We check to see if everything went ok. If the value in c is not equal to NULL we set the data variable in our structure to point to the newly allocated memory.

If there was a problem, we pop up an error message on the screen letting the user know that the program was unable to allocate memory for the texture buffer. NULL is returned.

```
if ( c != NULL ) {
ti->data = c;
}
else {
MessageBox(NULL,"Could Not Allocate Memory For A Texture Buffer","BUFFER ERROR",MB_OK |
MB_ICONINFORMATION);
return NULL;
}
}
```

If anything went wrong when we were trying to allocate memory for our image structure, the code below would pop up an error message and return NULL.

If there were no problems, we return ti which is a pointer to our newly allocated image structure. Whew... Hope that all made sense.

```
else
{
MessageBox(NULL,"Could Not Allocate An Image Structure","IMAGE STRUCTURE ERROR",MB_OK |
MB_ICONINFORMATION);
return NULL;
}
return ti; // Return Pointer To Image Struct
}
```

When it comes time to release the memory, the code below will deallocate the texture buffer and then free the image structure. t is a pointer to the TEXTURE_IMAGE data structure we want to deallocate.

```
// Free Up The Image Data
void DeallocateTexture( P_TEXTURE_IMAGE t )
{
if(t)
{
if(t->data)
{
free(t->data); // Free Its Image Buffer
}
free(t); // Free Itself
}
}
```

Now we read in our .RAW image. We pass the filename and a pointer to the image structure we want to load the image into. We set up our misc variables, and then calculate the size of a row. We figure out the size of a row by multiplying the width of our image by the format (bytes per pixel). So if the image was 256 pixels wide and there were 4 bytes per pixel, the width of a row would be 1024 bytes. We store the width of a row in stride.

We set up a pointer (p), and then attempt to open the file.

```
// Read A .RAW File In To The Allocated Image Buffer Using data In The Image Structure Header.
// Flip The Image Top To Bottom. Returns 0 For Failure Of Read, Or Number Of Bytes Read.
int ReadTextureData ( char *filename, P_TEXTURE_IMAGE buffer)
{
FILE *f;
int i,j,k,done=0;
int stride = buffer->width * buffer->format; // Size Of A Row (Width * Bytes Per Pixel)
unsigned char *p = NULL;

f = fopen(filename, "rb"); // Open "filename" For Reading Bytes
if( f != NULL ) // If File Exists
{
```

If the file exists, we set up the loops to read in our texture. i starts at the bottom of the image and moves up a line at a time. We start at the bottom so that the image is flipped the right way. .RAW images are stored upside down. We have to set our pointer now so that the data is loaded into the proper spot in the image buffer. Each time we move up a line (i is decreased) we set the pointer to the start of the new line. data is where our image buffer starts, and to move an entire line at a time in the buffer, multiply i by stride. Remember that stride is the length of a line in bytes, and i is the current line. So by multiplying the two, we move an entire line at a time.

The j loop moves from left (0) to right (width of line in pixels, not bytes).

```
for( i = buffer->height-1; i >= 0 ; i-- ) // Loop Through Height (Bottoms Up - Flip Image)
{
p = buffer->data + (i * stride );
for ( j = 0; j < buffer->width ; j++ ) // Loop Through Width
{
```

The k loop reads in our bytes per pixel. So if format (bytes per pixel) is 4, k loops from 0 to 2 which is bytes per pixel minus one (format-1). The reason we subtract one is because most raw images don't have an alpha value. We want to make the 4th byte our alpha value, and we want to set the alpha value manually.

Notice in the loop we also increase the pointer (p) and a variable called done. More about done later.

the line inside the loop reads a character from our file and stores it in the texture buffer at our current pointer location. If our image has 4 bytes per pixel, the first 3 bytes will be read from the .RAW file (format-1), and the 4th byte will be manually set to 255. After we set the 4th byte to 255 we increase the pointer location by one so that our 4th byte is not overwritten with the next byte in the file.

After a all of the bytes have been read in per pixel, and all of the pixels have been read in per row, and all of the rows have been read in, we are done! We can close the file.

```
for ( k = 0 ; k < buffer->format-1 ; k++, p++, done++ )
{
*p = fgetc(f); // Read Value From File And Store In Memory
}
*p = 255; p++; // Store 255 In Alpha Channel And Increase Pointer
}
}
fclose(f); // Close The File
}
```

If there was a problem opening the file (does not exist, etc), the code below will pop up a message box letting the user know that the file could not be opened.

The last thing we do is return done. If the file couldn't be opened, done will equal 0. If everything went ok, done should equal the number of bytes read from the file. Remember, we were increasing done every time we read a byte in the loop above (k loop).

```
else // Otherwise
{
MessageBox(NULL,"Unable To Open Image File","IMAGE ERROR",MB_OK | MB_ICONINFORMATION);
}
return done; // Returns Number Of Bytes Read In
}
```

This shouldn't need explaining. By now you should know how to build a texture. tex is the pointer to the TEXTURE_IMAGE structure that we want to use. We build a linear filtered texture. In this example, we're building mipmaps (smoother looking). We pass the width, height and data just like we would if we were using glaux, but this time we get the information from the selected TEXTURE_IMAGE structure.

```
void BuildTexture (P_TEXTURE_IMAGE tex)
{
glGenTextures(1, &texture[0]);
glBindTexture(GL_TEXTURE_2D, texture[0]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, tex->width, tex->height, GL_RGBA, GL_UNSIGNED_BYTE,
tex->data);
}
```

Now for the blitter code :) The blitter code is very powerful. It lets you copy any section of a (src) texture and paste it into a destination (dst) texture. You can combine as many textures as you want, you can set the alpha value used for blending, and you can select whether the two images blend together or cancel eachother out.

src is the TEXTURE_IMAGE structure to use as the source image. dst is the TEXTURE_IMAGE structure to use for the destination image. src_xstart is where you want to start copying from on the x axis of the source image. src_ystart is where you want to start copying from on the y axis of the source image. src_width is the width in pixels of the area you want to copy from the source image. src_height is the height in pixels of the area you want to copy from the source image. dst_xstart and dst_ystart is where you want to place the copied pixels from the source image onto the destination image. If blend is 1, the two images will be blended. alpha sets how tranparent the copied image will be when it mapped onto the destination image. 0 is completely clear, and 255 is solid.

We set up all our misc loop variables, along with pointers for our source image (s) and destination image (d). We check to see if the alpha value is within range. If not, we clamp it. We do the same for the blend value. If it's not 0-off or 1-on, we clamp it.

```
void Blit( P_TEXTURE_IMAGE src, P_TEXTURE_IMAGE dst, int src_xstart, int src_ystart, int
src_width, int src_height,
int dst_xstart, int dst_ystart, int blend, int alpha)
{
int i,j,k;
unsigned char *s, *d; // Source & Destination

// Clamp Alpha If Value Is Out Of Range
if( alpha > 255 ) alpha = 255;
if( alpha < 0 ) alpha = 0;

// Check For Incorrect Blend Flag Values
if( blend < 0 ) blend = 0;
if( blend > 1 ) blend = 1;
```

Now we have to set up the pointers. The destination pointer is the location of the destination data plus the starting location on the destination images y axis (dst_ystart) * the destination images width in pixels * the destination images bytes per pixel (format). This should give us the starting row for our destination image.

We do pretty much the same thing for the source pointer. The source pointer is the location of the source data plus the starting location on the source images y axis (src_ystart) * the source images width in pixels * the source images bytes per pixel (format). This should give us the starting row for our source image.

i loops from 0 to src_height which is the number of pixels to copy up and down from the source image.

```
d = dst->data + (dst_ystart * dst->width * dst->format); // Start Row - dst (Row * Width In
Pixels * Bytes Per Pixel)
s = src->data + (src_ystart * src->width * src->format); // Start Row - src (Row * Width In
Pixels * Bytes Per Pixel)

for (i = 0 ; i < src_height ; i++ ) // Height Loop
{
```

We already set the source and destination pointers to the correct rows in each image. Now we have to move to the correct location from left to right in each image before we can start blitting the data. We increase the location of the source pointer (s) by src_xstart which is the starting location on the x axis of the source image times the source images bytes per pixel. This moves the source (s) pointer to the starting pixel location on the x axis (from left to right) on the source image.

We do the exact same thing for the destination pointer. We increase the location of the destination pointer (d) by dst_xstart which is the starting location on the x axis of the destination image multiplied by the destination images bytes per pixel (format). This moves the destination (d) pointer to the starting pixel location on the x axis (from left to right) on the destination image.

After we have calculated where in memory we want to grab our pixels from (s) and where we want to move them to (d), we start the j loop. We'll use the j loop to travel from left to right through the source image.

```
s = s + (src_xstart * src->format); // Move Through Src Data By Bytes Per Pixel
d = d + (dst_xstart * dst->format); // Move Through Dst Data By Bytes Per Pixel
for (j = 0 ; j < src_width ; j++ ) // Width Loop
{
```

The k loop is used to go through all the bytes per pixel. Notice as k increases, our pointers for the source and destination images also increase.

Inside the loop we check to see if blending is on or off. If blend is 1, meaning we should blend, we do some fancy math to calculate the color of our blended pixels. The destination value (d) will equal our source value (s) multiplied by our alpha value + our current destination value (d) times 255 minus the alpha value. The shift operator (>>8) keeps the value in a 0-255 range.

If blending is disabled (0), we copy the data from the source image directly into the destination image. No blending is done and the alpha value is ignored.

```
for( k = 0 ; k < src->format ; k++, d++, s++) // "n" Bytes At A Time
{
if (blend) // If Blending Is On
*d = ( (*s * alpha) + (*d * (255-alpha)) ) >> 8; // Multiply Src Data*alpha Add Dst Data*(255-
alpha)
else // Keep in 0-255 Range With >> 8
*d = *s; // No Blending Just Do A Straight Copy
}
}
d = d + (dst->width - (src_width + dst_xstart))*dst->format; // Add End Of Row
s = s + (src->width - (src_width + src_xstart))*src->format; // Add End Of Row
}
}
```

The InitGL() code has changed quite a bit. All of the code below is new. We start off by allocating enough memory to hold a 256x256x4 Bytes Per Pixel Image. t1 will point to the allocated ram if everything went well.

After allocating memory for our image, we attempt to load the image. We pass ReadTextureData() the name of the file we wish to open, along with a pointer to our Image Structure (t1).

If we were unable to load the .RAW image, a message box will pop up on the screen to let the user know there was a problem loading the texture.

We then do the same thing for t2. We allocate memory, and attempt to read in our second .RAW image. If anything goes wrong we pop up a message box.

```
int InitGL(GLvoid) // This Will Be Called Right After The GL Window Is Created
{
t1 = AllocateTextureBuffer( 256, 256, 4 ); // Get An Image Structure
if (ReadTextureData("Data/Monitor.raw",t1)==0) // Fill The Image Structure With Data
{ // Nothing Read?
MessageBox(NULL,"Could Not Read 'Monitor.raw' Image Data","TEXTURE ERROR",MB_OK |
MB_ICONINFORMATION);
return FALSE;
}

t2 = AllocateTextureBuffer( 256, 256, 4 ); // Second Image Structure
if (ReadTextureData("Data/GL.raw",t2)==0) // Fill The Image Structure With Data
{ // Nothing Read?
MessageBox(NULL,"Could Not Read 'GL.raw' Image Data","TEXTURE ERROR",MB_OK |
MB_ICONINFORMATION);
return FALSE;
}
```

If we got this far, it's safe to assume the memory has been allocated and the images have been loaded. Now to use our Blit() command to merge the two images into one.

We start off by passing Blit() t2 and t1, both point to our TEXTURE_IMAGE structures (t2 is the second image, t1 is the first image.

Then we have to tell blit where to start grabbing data from on the source image. If you load the source image into Adobe Photoshop or any other program capable of loading .RAW images you will see that the entire image is blank except for the top right corner. The top right has a picture of the ball with GL written on it. The bottom left corner of the image is 0,0. The top right of the image is the width of the image-1 (255), the height of the image-1 (255). Knowing that we only want to copy 1/4 of the src image (top right), we tell Blit() to start grabbing from 127,127 (center of our source image).

Next we tell blit how many pixels we want to copy from our source point to the right, and from our source point up. We want to grab a 1/4 chunk of our image. Our image is 256x256 pixels, 1/4 of that is 128x128 pixels. All of the source information is done. Blit() now knows that it should copy from 127 on the x axis to 127+128 (255) on the x axis, and from 127 on the y axis to 127+128 (255) on the y axis.

So Blit() knows what to copy, and where to get the data from, but it doesn't know where to put the data once it's gotten it. We want to draw the ball with GL written on it in the middle our the monitor image. You find the center of the destination image (256x256) which is 128x128 and subtract half the width and height of the source image (128x128) which is 64x64. So (128-64) x (128-64) gives us a starting location of 64,64.

Last thing to do is tell our blitter routine we want to blend the two image (A one means blend, a zero means do not blend), and how much to blend the images. If the last value is 0, we blend the images 0%, meaning anything we copy will replace what was already there. If we use a value of 127, the two images blend together at 50%, and if you use 255, the image you are copying will be completely transparent and will not show up at all.

The pixels are copied from image2 (t2) to image1 (t1). The mixed image will be stored in t1.

```
// Image To Blend In, Original Image, Src Start X & Y, Src Width & Height, Dst Location X & Y,
Blend Flag, Alpha Value
Blit(t2,t1,127,127,128,128,64,64,1,127); // Call The Blitter Routine
```

After we have mixed the two images (t1 and t2) together, we build a texture from the combined images (t1).

After the texture has been created, we can deallocate the memory holding our two TEXTURE_IMAGE structures.

The rest of the code is pretty standard. We enable texture mapping, depth testing, etc.

```
BuildTexture (t1); // Load The Texture Map Into Texture Memory

DeallocateTexture( t1 ); // Clean Up Image Memory Because Texture Is
DeallocateTexture( t2 ); // In GL Texture Memory Now

glEnable(GL_TEXTURE_2D); // Enable Texture Mapping

glShadeModel(GL_SMOOTH); // Enables Smooth Color Shading
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // This Will Clear The Background Color To Black
glClearDepth(1.0); // Enables Clearing Of The Depth Buffer
glEnable(GL_DEPTH_TEST); // Enables Depth Testing
glDepthFunc(GL_LESS); // The Type Of Depth Test To Do

return TRUE;
}
```

I shouldn't even have to explain the code below. We move 5 units into the screen, select our single texture, and draw a texture mapped cube. You should notice that both textures are now combined into one. We don't have to render everything twice to map both textures onto the cube. The blitter code combined the images for us.

```
GLvoid DrawGLScene(GLvoid)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
glTranslatef(0.0f,0.0f,-5.0f);

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

glBindTexture(GL_TEXTURE_2D, texture[0]);

glBegin(GL_QUADS);
// Front Face
glNormal3f( 0.0f, 0.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Back Face
glNormal3f( 0.0f, 0.0f,-1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
// Top Face
glNormal3f( 0.0f, 1.0f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
// Bottom Face
glNormal3f( 0.0f,-1.0f, 0.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Right Face
glNormal3f( 1.0f, 0.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Left Face
glNormal3f(-1.0f, 0.0f, 0.0f);
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();

xrot+=0.3f;
yrot+=0.2f;
zrot+=0.4f;
}
```

The KillGLWindow() code has a few changes. You'll notice the code to switch from fullscreen mode back to your desktop is now at the top of KillGLWindow(). If the user ran the program in fullscreen mode, the first thing we do when we kill the window is try to switch back to the desktop resolution. If the quick way fails to work, we reset the screen using the information stored in DMsaved. This should restore us to our orignal desktop settings.

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
{
if (fullscreen) // Are We In Fullscreen Mode?
{
if (!ChangeDisplaySettings(NULL,CDS_TEST)) { // If The Shortcut Doesn't Work
ChangeDisplaySettings(NULL,CDS_RESET); // Do It Anyway (To Get The Values Out Of The Registry)
ChangeDisplaySettings(&DMsaved,CDS_RESET); // Change Resolution To The Saved Settings
}
else // Not Fullscreen
{
ChangeDisplaySettings(NULL,CDS_RESET); // Do Nothing
}

ShowCursor(TRUE); // Show Mouse Pointer
}

if (hRC) // Do We Have A Rendering Context?
{
if (!wglMakeCurrent(NULL,NULL)) // Are We Able To Release The DC And RC Contexts?
{
MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
}

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

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

if (hWnd && !DestroyWindow(hWnd)) // Are We Able To Destroy The Window?
{
MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
hWnd=NULL; // Set hWnd To NULL
}

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

I've made some changes in CreateGLWindow. The changes will hopefully elimintate alot of the problems people are having when they switch to and from from fullscreen mode. I've included the first part of CreateGLWindow() so you can easily follow through the code.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
GLuint PixelFormat; // Holds The Results After Searching For A Match
WNDCLASS wc; // Windows Class Structure
DWORD dwExStyle; // Window Extended Style
DWORD dwStyle; // Window Style

fullscreen=fullscreenflag; // Set The Global Fullscreen Flag

hInstance = GetModuleHandle(NULL); // Grab An Instance For Our Window
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // Redraw On Size, And Own DC For Window.
wc.lpfnWndProc = (WNDPROC) WndProc; // WndProc Handles Messages
wc.cbClsExtra = 0; // No Extra Window Data
wc.cbWndExtra = 0; // No Extra Window Data
wc.hInstance = hInstance; // Set The Instance
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Load The Default Icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Load The Arrow Pointer
wc.hbrBackground = NULL; // No Background Required For GL
wc.lpszMenuName = NULL; // We Don't Want A Menu
wc.lpszClassName = "OpenGL"; // Set The Class Name
```

The big change here is that we now save the current desktop resolution, bit depth, etc. before we switch to fullscreen mode. That way when we exit the program, we can set everything back exactly how it was. The first line below copies the display settings into the DMsaved Device Mode structure. Nothing else has changed, just one new line of code.

```
EnumDisplaySettings(NULL, ENUM_CURRENT_SETTINGS, &DMsaved); // Save The Current Display State
(NEW)

if (fullscreen) // Attempt Fullscreen Mode?
{
```

```
DEVMODE dmScreenSettings; // Device Mode
memset(&dmScreenSettings,0,sizeof(dmScreenSettings)); // Makes Sure Memory's Cleared
dmScreenSettings.dmSize=sizeof(dmScreenSettings); // Size Of The Devmode Structure
dmScreenSettings.dmPelsWidth = width; // Selected Screen Width
dmScreenSettings.dmPelsHeight = height; // Selected Screen Height
dmScreenSettings.dmBitsPerPel = bits; // Selected Bits Per Pixel
dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

// Try To Set Selected Mode And Get Results. NOTE: CDS_FULLSCREEN Gets Rid Of Start Bar.
if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
{
// If The Mode Fails, Offer Two Options. Quit Or Use Windowed Mode.
if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use
Windowed Mode Instead?","NeHe GL",MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
{
fullscreen=FALSE; // Windowed Mode Selected. Fullscreen = FALSE
}
else
{
// Pop Up A Message Box Letting User Know The Program Is Closing.
MessageBox(NULL,"Program Will Now Close.","ERROR",MB_OK|MB_ICONSTOP);
return FALSE; // Return FALSE
}
}
}
```

WinMain() starts out the same as always. Ask the user if they want fullscreen or not, then start the loop.

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}

// Create Our OpenGL Window
if (!CreateGLWindow("Andreas Löffler, Rob Fletcher & NeHe's Blitter & Raw Image Loading
Tutorial", 640, 480, 32, fullscreen))
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
```

I have made some changes to the code below. If the program is not active (minimized) we wait for a message with the command WaitMessage(). Everything stops until the program receives a message (usually maximizing the window). What this means is that the program no longer hogs the processor while it's minimized. Thanks to Jim Strong for the suggestion.

```
if (!active) // Program Inactive?
{
WaitMessage(); // Wait For A Message / Do Nothing ( NEW ... Thanks Jim Strong )
}

if (keys[VK_ESCAPE]) // Was Escape Pressed?
{
done=TRUE; // ESC Signalled A Quit
}

if (keys[VK_F1]) // Is F1 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("Andreas Löffler, Rob Fletcher & NeHe's Blitter & Raw Image Loading
Tutorial",640,480,16,fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}

DrawGLScene(); // Draw The Scene
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
}

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

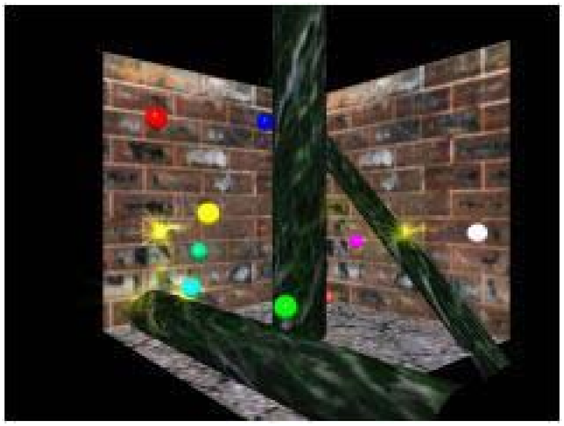
Well, that´s it! Now the doors are open for creating some very cool blending effects for your games, engines or even applications. With texture buffers we used in this tutorial you could do more cool effects like real-time plasma or water. When combining these effects all together you´re able to do nearly photo-realistic terrain. If something doesn´t work in this tutorial or you have suggestions how to do it better, then please don´t hesitate to E-Mail me. Thank you for reading and good luck in creating your own special effects!

Some information about Andreas: I´m an 18 years old pupil who is currently studying to be a software engineer. I´ve been programming for nearly 10 years now. I've been programming in OpenGL for about 1.5 years.

Andreas Löffler & **Rob Fletcher**

Jeff Molofee (**NeHe**)

Lesson 30
Collision Detection



Collision Detection and Physically Based Modeling Tutorial by Dimitrios Christopoulos (christop@fhw.gr).

The source code upon which this tutorial is based, is from an older contest entry of mine (at OGLchallenge.dhs.org). The theme was Collision Crazy and my entry (which by the way took the 1st place :)) was called Magic Room. It features collision detection, physically based modeling and effects.

Collision Detection

A difficult subject and to be honest as far as I have seen up until now, there has been no easy solution for it. For every application there is a different way of finding and testing for collisions. Of course there are brute force algorithms which are very general and would work with any kind of objects, but they are expensive.

We are going to investigate algorithms which are very fast, easy to understand and to some extent quite flexible. Furthermore importance must be given on what to do once a collision is detected and how to move the objects, in accordance to the laws of physics. We have a lot stuff to cover. Lets review what we are going to learn:

1) Collision Detection

- Moving Sphere - Plane
- Moving Sphere - Cylinder
- Moving Sphere - Moving Sphere

2) Physically Based Modeling

- Collision Response
- Moving Under Gravity Using Euler Equations

3) Special Effects

- Explosion Modeling Using A Fin-Tree Billboard Method
- Sounds Using The Windows Multimedia Library (Windows Only)

4) Explanation Of The Code

- The Code Is Divided Into 5 Files

Lesson30.cpp		: Main Code For This Tutorial
Image.cpp,	Image.h	: Code To Load Bitmaps
Tmatrix.cpp,	Tmatrix.h	: Classes To Handle Rotations
Tray.cpp,	Tray.h	: Classes To Handle Ray Operations
Tvector.cpp,	Tvector.h	: Classes To Handle Vector Operations

A lot of handy code! The Vector, Ray and Matrix classes are very useful. I used them until now for personal projects of my own.

1) Collision Detection

For the collision detection we are going to use algorithms which are mostly used in ray tracing. Lets first define a ray.

A ray using vector representation is represented using a vector which denotes the start and a vector (usually normalized) which is the direction in which the ray travels. Essentially a ray starts from the start point and travels in the direction of the direction vector. So our ray equation is:

*PointOnRay = Raystart + t * Raydirection*

t is a float which takes values from [0, infinity).

With 0 we get the start point and substituting other values we get the corresponding points along the ray. PointOnRay, Raystart, Raydirection, are 3D Vectors with values (x,y,z). Now we can use this ray representation and calculate the intersections with plane or cylinders.

Ray - Plane Intersection Detection

A plane is represented using its Vector representation as:

Xn dot X = d

Xn, X are vectors and d is a floating point value.
Xn is its normal.
X is a point on its surface.
d is a float representing the distance of the plane along the normal, from the center of the coordinate system.

Essentially a plane represents a half space. So all that we need to define a plane is a 3D point and a normal from that point which is perpendicular to that plane. These two vectors form a plane, ie. if we take for the 3D point the vector (0,0,0) and for the normal (0,1,0) we essentially define a plane across x,z axes. Therefore defining a point and a normal is enough to compute the Vector representation of a plane.

Using the vector equation of the plane the normal is substituted as Xn and the 3D point from which the normal originates is substituted as X. The only value that is missing is d which can easily be computed using a dot product (from the vector equation).

(Note: This Vector representation is equivalent to the widely known parametric form of the plane Ax + By + Cz + D=0 just take the three x,y,z values of the normal as A,B,C and set D=-d).

The two equations we have so far are:

*PointOnRay = Raystart + t * Raydirection*
Xn dot X = d

If a ray intersects the plane at some point then there must be some point on the ray which satisfies the plane equation as follows:

Xn dot PointOnRay = d or *(Xn dot Raystart) + t * (Xn dot Raydirection) = d*

solving for t:

t = (d - Xn dot Raystart) / (Xn dot Raydirection)

replacing d:

t= (Xn dot PointOnRay - Xn dot Raystart) / (Xn dot Raydirection)

summing it up:

t= (Xn dot (PointOnRay - Raystart)) / (Xn dot Raydirection)

t represents the distance from the start until the intersection point along the direction of the ray. Therefore substituting t into the ray equation we can get the collision point. There are a few special cases though. If Xn dot Raydirection = 0 then these two vectors are perpendicular (ray runs parallel to plane) and there will be no collision. If t is negative the collision takes place behind the starting point of the ray along the opposite direction and again there is no intersection.

```
int TestIntersionPlane(const Plane& plane,const TVector& position,const TVector& direction,
double& lamda, TVector& pNormal)
{
double DotProduct=direction.dot(plane._Normal); // Dot Product Between Plane Normal And Ray
Direction
double l2;

// Determine If Ray Parallel To Plane
if ((DotProduct<ZERO)&&(DotProduct>-ZERO))
return 0;

l2=(plane._Normal.dot(plane._Position-position))/DotProduct; // Find Distance To Collision Point

if (l2<-ZERO) // Test If Collision Behind Start
return 0;

pNormal=plane._Normal;
lamda=l2;
return 1;
}
```

The code above calculates and returns the intersection. It returns 1 if there is an intersection otherwise it returns 0. The parameters are the plane, the start and direction of the vector, a double (lamda) where the collision distance is stored if there was any, and the returned normal at the collision point.

Ray - Cylinder Intersection

Computing the intersection between an infinite cylinder and a ray is much more complicated that is why I won't explain it here. There is way too much math involved too easily explain and my goal is primarily to give you tools how to do it without getting into alot of detail (this is not a geometry class). If anyone is interested in the theory behind the intersection code, please look at the Graphic Gems II Book (pp 35, intersection of a with a cylinder). A cylinder is represented as a ray, using a start and direction (here it represents the axis) vector and a radius (radius around the axis of the cylinder). The relevant function is:

```
int TestIntersionCylinder(const Cylinder& cylinder,const TVector& position,const TVector&
direction, double& lamda, TVector& pNormal,TVector& newposition)
```

Returns 1 if an intersection was found and 0 otherwise.

The parameters are the cylinder structure (look at the code explanation further down), the start, direction vectors of the ray. The values returned through the parameters are the distance, the normal at the intersection point and the intersection point itself.

Sphere - Sphere Collision

A sphere is represented using its center and its radius. Determining if two spheres collide is easy. By finding the distance between the two centers (dist method of the TVector class) we can determine if they intersect, if the distance is less than the sum of their two radius.

The problem lies in determining if 2 MOVING spheres collide. Bellow is an example where 2 sphere move during a time step from one point to another. Their paths cross in-between but this is not enough to prove that an intersection occurred (they could pass at a different time) nor can the collision point be determined.

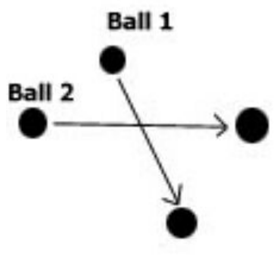


Figure 1

The previous intersection methods were solving the equations of the objects to determine the intersection. When using complex shapes or when these equations are not available or can not be solved, a different method has to be used. The start points, endpoints, time step, velocity (direction of the sphere + speed) of the sphere and a method of how to compute intersections of static spheres is already known. To compute the intersection, the time step has to be sliced up into smaller pieces. Then we move the spheres according to that sliced time step using its velocity, and check for collisions. If at any point collision is found (which means the spheres have already penetrated each other) then we take the previous position as the intersection point (we could start interpolating between these points to find the exact intersection position, but that is mostly not required).

The smaller the time steps, the more slices we use the more accurate the method is. As an example lets say the time step is 1 and our slices are 3. We would check the two balls for collision at time 0 , 0.33, 0.66, 1. Easy !!!!

The code which performs this is:

```
/*****************************************************************************************/
/*** Find if any of the current balls ***/
/*** intersect with each other in the current timestep ***/
/*** Returns the index of the 2 intersecting balls, the point and time of intersection ***/
/*****************************************************************************************/

int FindBallCol(TVector& point, double& TimePoint, double Time2, int& BallNr1, int& BallNr2)
{
TVector RelativeV;
TRay rays;
double MyTime=0.0, Add=Time2/150.0, Timedummy=10000, Timedummy2=-1;
TVector posi; // Test All Balls Against Eachother In 150 Small Steps
for (int i=0;i<NrOfBalls-1;i++)
{
for (int j=i+1;j<NrOfBalls;j++)
{
RelativeV=ArrayVel[i]-ArrayVel[j]; // Find Distance
rays=TRay(OldPos[i],TVector::unit(RelativeV));
MyTime=0.0;

if ( (rays.dist(OldPos[j])) > 40) continue; // If Distance Between Centers Greater Than 2*radius
// An Intersection Occurred
while (MyTime<Time2) // Loop To Find The Exact Intersection Point
{
MyTime+=Add;
posi=OldPos[i]+RelativeV*MyTime;
if (posi.dist(OldPos[j])<=40)
{
point=posi;
if (Timedummy>(MyTime-Add)) Timedummy=MyTime-Add;
BallNr1=i;
BallNr2=j;
break;
}
}
}
}

if (Timedummy!=10000)
{
TimePoint=Timedummy;
return 1;
}
return 0;
}
```

How To Use What We Just Learned

So now that we can determine the intersection point between a ray and a plane/cylinder we have to use it somehow to determine the collision between a sphere and one of these primitives. What we can do so far is determine the exact collision point between a particle and a plane/cylinder. The start position of the ray is the position of the particle and the direction of the ray is its velocity (speed and direction). To make it usable for spheres is quite easy. Look at Figure 2a to see how this can be accomplished.

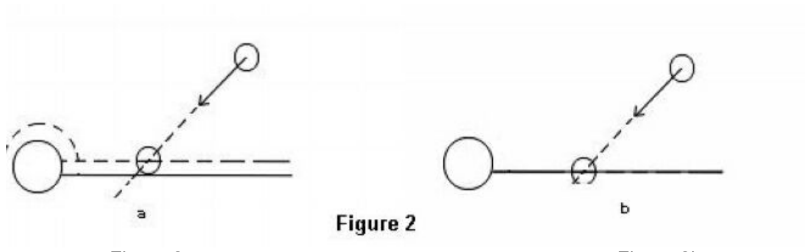


Figure 2a Figure 2b

Each sphere has a radius, take the center of the sphere as the particle and offset the surface along the normal of each plane/cylinder of interest. In Figure 2a these new primitives are represented with dotted lines. Your actual primitives of interest are the ones represented by continuous lines, but the collision testing is done with the offset primitives (represented with dotted lines). In essence we perform the intersection test with a little offset plane and a larger in radius cylinder. Using this little trick the ball does not penetrate the surface if an intersection is determined with its center. Otherwise we get a situation as in Figure 2b, where be sphere penetrates the surface. This happens because we determine the intersection between its center and the primitives, which means we did not modify our original code!

Having determined where the collision takes place we have to determine if the intersection takes place in our current time step. Timestep is the time we move our sphere from its current point according to its velocity. Because we are testing with infinite rays there is always the possibility that the collision point is after the new position of the sphere. To determine this we move the sphere, calculate its new position and find the distance between the start and end point. From our collision detection procedure we also get the distance from the start point to its collision point. If this distance is less than the distance between start and end point then there is a collision. To calculate the exact time we solve the following simple equation. Represent the distance between start - end point with Dst, the distance between start - collision point Dsc, and the time step as T. The time where the collision takes place (Tc) is:

Tc= Dsc*T / Dst

All this is performed of course if an intersection is determined. The returned time is a fraction of the whole time step, so if the time step was 1 sec, and we found an intersection exactly in the middle of the distance, the calculated collision time would be 0.5 sec. this is interpreted as "0.5 sec after the start there is an intersection". Now the intersection point can be calculated by just multiplying Tc with the current velocity and adding it to the start point.

Collision point= Start + Velocity*Tc

This is the collision point on the offset primitive, to find the collision point on the real primitive we add to that point the reverse of the normal at that point (which is also returned by the intersection routines) by the radius of the sphere. Note that the cylinder intersection routine returns the intersection point if there is one so it does not need to be calculated.

2) Physically Based Modeling

Collision Response

To determine how to respond after hitting Static Objects like Planes, Cylinders is as important as finding the collision point itself. Using the algorithms and functions described, the exact collision point, the normal at the collision point and the time within a time step in which the collision occurs can be found.

To determine how to respond to a collision, laws of physics have to be applied. When an object collides with the surface its direction changes i.e.. it bounces off. The angle of the of the new direction (or reflection vector) with the normal at the collision point is the same as the original direction vector. Figure 3 shows a collision with a sphere.

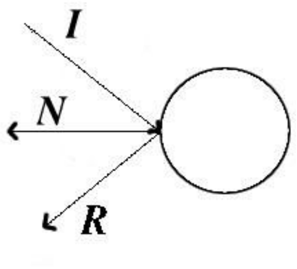


Figure 3

R is the new direction vector
I is the old direction vector before the collision
N is the Normal at the collision point

The new vector R is calculated as follows:

R= 2*(-I dot N)*N + I

The restriction is that the I and N vectors have to be unit vectors. The velocity vector as used in our examples represents speed and direction. Therefore it can not be plugged into the equation in the place of I, without any transformation. The speed has to be extracted. The speed for such a velocity vector is extracted finding the magnitude of the vector. Once the magnitude is found, the vector can be transformed to a unit vector and plugged into the equation giving the reflection vector R. R shows us now the direction, of the reflected ray, but in order to be used as a velocity vector it must also incorporate the speed. Therefore it gets, multiplied with the magnitude of the original ray, thus resulting in the correct velocity vector.

In the example this procedure is applied to compute the collision response if a ball hits a plane or a cylinder. But it works also for arbitrary surfaces, it does not matter what the shape of the surface is. As long as a collision point and a Normal can be found the collision response method is always the same. The code which does these operations is:

```
rt2=ArrayVel[BallNr].mag(); // Find Magnitude Of Velocity
ArrayVel[BallNr].unit(); // Normalize It
```

```
// Compute Reflection
ArrayVel[BallNr]=TVector::unit( (normal*(2*normal.dot(-ArrayVel[BallNr]))) + ArrayVel[BallNr] );
ArrayVel[BallNr]=ArrayVel[BallNr]*rt2; // Muliply With Magnitude To Obtain Final Velocity Vector
```

When Spheres Hit Other Spheres

Determining the collision response, if two balls hit each other is much more difficult. Complex equations of particle dynamics have to be solved and therefore I will just post the final solution without any proof. Just trust me on this one :) During the collision of 2 balls we have a situation as it is depicted in Figure 4.

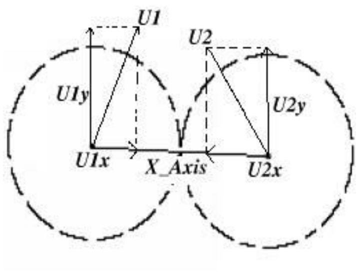


Figure 4

U1 and U2 are the velocity vectors of the two spheres at the time of impact. There is an axis (X_Axis) vector which joins the 2 centers of the spheres, and U1x, U2x are the projected vectors of the velocity vectors U1,U2 onto the axis (X_Axis) vector.

U1y and U2y are the projected vectors of the velocity vectors U1,U2 onto the axis which is perpendicular to the X_Axis. To find these vectors a few simple dot products are needed. M1, M2 is the mass of the two spheres respectively. V1,V2 are the new velocities after the impact, and V1x, V1y, V2x, V2y are the projections of the velocity vectors onto the X_Axis.

In More Detail:

a) Find X_Axis

X_Axis = (center2 - center1);
Unify X_Axis, X_Axis.unit();

b) Find Projections

U1x= X_Axis * (X_Axis dot U1)
U1y= U1 - U1x
U2x =-X_Axis * (-X_Axis dot U2)
U2y =U2 - U2x

c)Find New Velocities

(U1x * M1)+(U2x*M2)-(U1x-U2x)*M2
V1x= --------------------------------
M1+M2
(U1x * M1)+(U2x*M2)-(U2x-U1x)*M1
V2x= --------------------------------
M1+M2

In our application we set the M1=M2=1, so the equations get even simpler.

d)Find The Final Velocities

V1y=U1y
V2y=U2y
V1=V1x+V1y
V2=V2x+V2y

The derivation of that equations has a lot of work, but once they are in a form like the above they can be used quite easily. The code which does the actual collision response is:

```
TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
double a,b;
pb1=OldPos[BallColNr1]+ArrayVel[BallColNr1]*BallTime; // Find Position Of Ball1
pb2=OldPos[BallColNr2]+ArrayVel[BallColNr2]*BallTime; // Find Position Of Ball2
xaxis=(pb2-pb1).unit(); // Find X-Axis
```

```
a=xaxis.dot(ArrayVel[BallColNr1]); // Find Projection
U1x=xaxis*a; // Find Projected Vectors
U1y=ArrayVel[BallColNr1]-U1x;
xaxis=(pb1-pb2).unit(); // Do The Same As Above
b=xaxis.dot(ArrayVel[BallColNr2]); // To Find Projection
U2x=xaxis*b; // Vectors For The Other Ball
U2y=ArrayVel[BallColNr2]-U2x;
V1x=(U1x+U2x-(U1x-U2x))*0.5; // Now Find New Velocities
V2x=(U1x+U2x-(U2x-U1x))*0.5;
V1y=U1y;
V2y=U2y;
for (j=0;j<NrOfBalls;j++) // Update All Ball Positions
ArrayPos[j]=OldPos[j]+ArrayVel[j]*BallTime;
ArrayVel[BallColNr1]=V1x+V1y; // Set New Velocity Vectors
ArrayVel[BallColNr2]=V2x+V2y; // To The Colliding Balls
```

Moving Under Gravity Using Euler Equations

To simulate realistic movement with collisions, determining the the collision point and computing the response is not enough. Movement based upon physical laws also has to be simulated.

The most widely used method for doing this is using Euler equations. As indicated all the computations are going to be performed using time steps. This means that the whole simulation is advanced in certain time steps during which all the movement, collision and response tests are performed. As an example we can advanced a simulation 2 sec. on each frame. Based on Euler equations, the velocity and position at each new time step is computed as follows:

Velocity_New = Velovity_Old + Acceleration*TimeStep
Position_New = Position_Old + Velocity_New*TimeStep

Now the objects are moved and tested angainst collision using this new velocity. The Acceleration for each object is determined by accumulating the forces which are acted upon it and divide by its mass according to this equation:

Force = mass * acceleration

A lot of physics formulas :)

But in our case the only force the objects get is the gravity, which can be represented right away as a vector indicating acceleration. In our case something negative in the Y direction like (0,-0.5,0). This means that at the beginning of each time step, we calculate the new velocity of each sphere and move them testing for collisions. If a collision occurs during a time step (say after 0.5 sec with a time step equal to 1 sec.) we advance the object to this position, compute the reflection (new velocity vector) and move the object for the remaining time (which is 0.5 in our example) testing again for collisions during this time. This procedure gets repeated until the time step is completed.

When multiple moving objects are present, each moving object is tested with the static geometry for intersections and the nearest intersection is recorded. Then the intersection test is performed for collisions among moving objects, where each object is tested with everyone else. The returned intersection is compared with the intersection returned by the static objects and the closest one is taken. The whole simulation is updated to that point, (i.e. if the closest intersection would be after 0.5 sec. we would move all the objects for 0.5 seconds), the reflection vector is calculated for the colliding object and the loop is run again for the remaining time.

3) Special Effects

Explosions

Every time a collision takes place an explosion is triggered at the collision point. A nice way to model explosions is to alpha blend two polygons which are perpendicular to each other and have as the center the point of interest (here intersection point). The polygons are scaled and disappear over time. The disappearing is done by changing the alpha values of the vertices from 1 to 0, over time. Because a lot of alpha blended polygons can cause problems and overlap each other (as it is stated in the Red Book in the chapter about transparency and blending) because of the Z buffer, we borrow a technique used in particle rendering. To be correct we had to sort the polygons from back to front according to their eye point distance, but disabling the Depth buffer writes (not reads) also does the trick (this is also documented in the red book). Notice that we limit our number of explosions to maximum 20 per frame, if additional explosions occur and the buffer is full, the explosion is discarded. The source which updates and renders the explosions is:

```
// Render / Blend Explosions
glEnable(GL_BLEND); // Enable Blending
glDepthMask(GL_FALSE); // Disable Depth Buffer Writes
glBindTexture(GL_TEXTURE_2D, texture[1]); // Upload Texture
for(i=0; i<20; i++) // Update And Render Explosions
{
if(ExplosionArray[i]._Alpha>=0)
{
glPushMatrix();
ExplosionArray[i]._Alpha-=0.01f; // Update Alpha
ExplosionArray[i]._Scale+=0.03f; // Update Scale
// Assign Vertices Colour Yellow With Alpha
// Colour Tracks Ambient And Diffuse
glColor4f(1,1,0,ExplosionArray[i]._Alpha); // Scale
glScalef(ExplosionArray[i]._Scale,ExplosionArray[i]._Scale,ExplosionArray[i]._Scale);
// Translate Into Position Taking Into Account The Offset Caused By The Scale
glTranslatef((float)ExplosionArray[i]._Position.X()/ExplosionArray[i]._Scale,
(float)ExplosionArray[i]._Position.Y()/ExplosionArray[i]._Scale,
(float)ExplosionArray[i]._Position.Z()/ExplosionArray[i]._Scale);
glCallList(dlist); // Call Display List
glPopMatrix();
}
}
```

Sound

For the sound the windows multimedia function PlaySound() is used. This is a quick and dirty way to play wav files quickly and without trouble.

4) Explaining the Code

Congratulations...

If you are still with me you have survived successfully the theory section ;) Before having fun playing around with the demo, some further explanations about the source code are necessary. The main flow and steps of the simulation are as follows (in pseudo code):

```
While (Timestep!=0)
{
For each ball
{
compute nearest collision with planes;
compute nearest collision with cylinders;
Save and replace if it the nearest intersection;
in time computed until now;
}
Check for collision among moving balls;
Save and replace if it the nearest intersection;
in time computed until now;
If (Collision occurred)
{
Move All Balls for time equal to collision time;
(We already have computed the point, normal and collision time.)
Compute Response;
Timestep-=CollisonTime;
}
else
Move All Balls for time equal to Timestep
}
```

The actual code which implements the above pseudo code is much harder to read but essentially is an exact implementation of the pseudo code above.

```
// While Time Step Not Over
while (RestTime>ZERO)
{
lamda=10000; // Initialize To Very Large Value
// For All The Balls Find Closest Intersection Between Balls And Planes / Cylinders
for (int i=0;i<NrOfBalls;i++)
{
// Compute New Position And Distance
OldPos[i]=ArrayPos[i];
TVector::unit(ArrayVel[i],uveloc);
ArrayPos[i]=ArrayPos[i]+ArrayVel[i]*RestTime;
rt2=OldPos[i].dist(ArrayPos[i]);
// Test If Collision Occured Between Ball And All 5 Planes
if (TestIntersionPlane(pl1,OldPos[i],uveloc,rt,norm))
{
// Find Intersection Time
rt4=rt*RestTime/rt2;
// If Smaller Than The One Already Stored Replace In Timestep
if (rt4<=lamda)
{
// If Intersection Time In Current Time Step
if (rt4<=RestTime+ZERO)
if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
{
normal=norm;
point=OldPos[i]+uveloc*rt;
lamda=rt4;
BallNr=i;
}
}
}
```

```
if (TestIntersionPlane(pl2,OldPos[i],uveloc,rt,norm))
{

// ...The Same As Above Omitted For Space Reasons
}

if (TestIntersionPlane(pl3,OldPos[i],uveloc,rt,norm))
{

// ...The Same As Above Omitted For Space Reasons
}

if (TestIntersionPlane(pl4,OldPos[i],uveloc,rt,norm))
{

// ...The Same As Above Omitted For Space Reasons
}

if (TestIntersionPlane(pl5,OldPos[i],uveloc,rt,norm))
{

// ...The Same As Above Omitted For Space Reasons
}

// Now Test Intersection With The 3 Cylinders
if (TestIntersionCylinder(cyl1,OldPos[i],uveloc,rt,norm,Nc))
{
rt4=rt*RestTime/rt2;
if (rt4<=lamda)
{
if (rt4<=RestTime+ZERO)
if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
{
normal=norm;
point=Nc;
lamda=rt4;
BallNr=i;
}
}
}

if (TestIntersionCylinder(cyl2,OldPos[i],uveloc,rt,norm,Nc))
{
// ...The Same As Above Omitted For Space Reasons
}

if (TestIntersionCylinder(cyl3,OldPos[i],uveloc,rt,norm,Nc))
{
// ...The Same As Above Omitted For Space Reasons
}

}

// After All Balls Were Tested With Planes / Cylinders Test For Collision
// Between Them And Replace If Collision Time Smaller
if (FindBallCol(Pos2,BallTime,RestTime,BallColNr1,BallColNr2))
{
if (sounds)
PlaySound("Explode.wav",NULL,SND_FILENAME|SND_ASYNC);

if ( (lamda==10000) || (lamda>BallTime) )
{
RestTime=RestTime-BallTime;
TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
double a,b;
.
.
Code Omitted For Space Reasons
The Code Is Described In The Physically Based Modeling
Section Under Sphere To Sphere Collision
.
.
//Update Explosion Array And Insert Explosion
for(j=0;j<20;j++)
{
if (ExplosionArray[j]._Alpha<=0)
{
ExplosionArray[j]._Alpha=1;
ExplosionArray[j]._Position=ArrayPos[BallColNr1];
ExplosionArray[j]._Scale=1;
break;
}
}

continue;
}
}

// End Of Tests
// If Collision Occured Move Simulation For The Correct Timestep
// And Compute Response For The Colliding Ball
if (lamda!=10000)
{
RestTime-=lamda;
for (j=0;j<NrOfBalls;j++)
ArrayPos[j]=OldPos[j]+ArrayVel[j]*lamda;
rt2=ArrayVel[BallNr].mag();
ArrayVel[BallNr].unit();
ArrayVel[BallNr]=TVector::unit( (normal*(2*normal.dot(-ArrayVel[BallNr]))) + ArrayVel[BallNr] );
ArrayVel[BallNr]=ArrayVel[BallNr]*rt2;

// Update Explosion Array And Insert Explosion
for(j=0;j<20;j++)
{
if (ExplosionArray[j]._Alpha<=0)
{
ExplosionArray[j]._Alpha=1;
ExplosionArray[j]._Position=point;
ExplosionArray[j]._Scale=1;
break;
}
}
}
else RestTime=0;
} // End Of While Loop
```

The Main Global Variables Of Importance Are:

Represent the direction and position of the camera. The camera is moved using the LookAt function. As you will probably notice, if not in hook mode (which I will explain later), the whole scene rotates around, the degree of rotation is handled with camera_rotation.	TVector dir
	TVector pos(0,-50,1000);
	float camera_rotation=0;

Represent the acceleration applied to the moving balls. Acts as gravity in the application.	TVector accel(0,-0.05,0);

Arrays which hold the New and old ball positions and the velocity vector of each ball. The number of balls is hard coded to 10.	TVector ArrayVel[10];
	TVector ArrayPos[10];
	TVector OldPos[10];
	int NrOfBalls=3;

The time step we use.	double Time=0.6;
If 1 the camera view changes and a (the ball with index 0 in the array) ball is followed. For making the camera following the ball we used its position and velocity vector to position the camera exactly behind the ball and make it look along the velocity vector of the ball.	int hook_toball1=0;
Self explanatory structures for holding data about explosions, planes and cylinders.	struct Plane struct Cylinder struct Explosion
The explosions are stored in a array, of fixed length.	Explosion ExplosionArray[20];

The Main Functions Of Interest Are:

Perform Intersection tests with primitives	int TestIntersionPlane(....);
	int TestIntersionCylinder(...);

Loads Textures from bmp files	void LoadGLTextures();
Has the rendering code. Renders the balls, walls, columns and explosions	void DrawGLScene();
Performs the main simulation logic	void idle();
Sets Up OpenGL state	void InitGL();
Find if any balls collide again each other in current time step	int FindBallCol(...);

For more information look at the source code. I tried to comment it as best as I could. Once the collision detection and response logic is understood, the source should become very clear. For any more info don't hesitate to contact me.

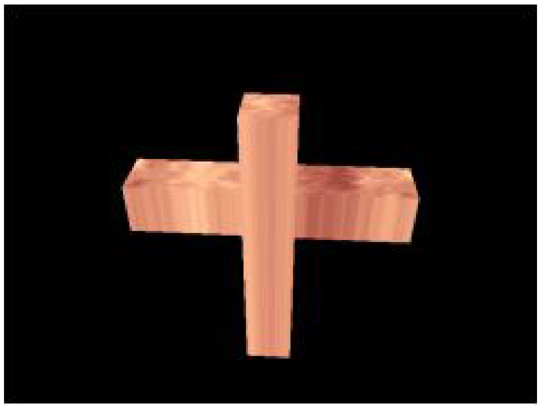
As I stated at the beginning of this tutorial, the subject of collision detection is a very difficult subject to cover in one tutorial. You will learn a lot in this tutorial, enough to create some pretty impressive demos of your own, but there is still alot more to learn on this subject. Now that you have the basics, all the other sources on Collision Detection and Physically Based Modeling out there should become easier to understand. With this said, I send you on your way and wish you happy collisions!!!

Some information about Dimitrios Christopoulos: He is currently working as a Virtual Reality software engineer at the Foundation of the Hellenic World in Athens/Greece (www.fhw.gr). Although Born in Germany, he studied in Greece at the University of Patras for a B.Sc. in Computer Engineering and Informatics. He holds also a MSc degree (honours) from the University of Hull (UK) in Computer Graphics and Virtual Environments. He did his first steps in game programming using Basic on an Commodore 64, and switched to C/C++/Assembly on the PC platform after the start of his studium. During the last few years OpenGL has become his graphics API of choice. For more information visit his site at: http://members.xoom.com/D_Christop.

Dimitrios Christopoulos

Jeff Molofee (**NeHe**)

Lesson 31 Model Loading



Model Rendering Tutorial by Brett Porter (brettporter@yahoo.com)

The source for this project has been extracted from PortaLib3D, a library I have written to enable users to do things like displaying models with very little extra code. But so that you can trust such a library, you should understand what it is doing, so this tutorial aims to help with that.

The portions of PortaLib3D included here retain my copyright notices. This doesn't mean they can't be used by you - it means that if you cut-and-paste the code into your project, you have to give me proper credit. That's all. If you choose to read, understand, and re-implement the code yourself (and it is what you are encouraged to do if you are not actually using the library. You don't learn anything with cut-and-paste!), then you free yourself of that obligation. Let's face it, the code is nothing special. Ok, let's get onto something more interesting!

OpenGL Base Code

The OpenGL base code is in Lesson32.cpp. Mostly it came from Lesson 6, with a small modification to the loading of textures and the drawing routine. The changes will be discussed later.

Milkshape 3D

The model I use in this example is from Milkshape 3D. The reason I use this is because it is a damn fine modelling package, and it includes its file-format so it is easy to parse and understand. My next plan is to implement an Anim8or (http://www.anim8or.com) file reader because it is free and of course a 3DS reader.

However, the file format, while it will be described briefly here, is not the major concern for loading a model. You must create your own structures that are suitable to store the data, and then read the file into that. So first, let's describe the structures required for a model.

Model Data Structures

These model data structures come from the class Model in Model.h. First, and most important, we need vertices:

```
// Vertex Structure
struct Vertex
{
char m_boneID; // For Skeletal Animation
float m_location[3];
};

// Vertices Used
int m_numVertices;
Vertex *m_pVertices;
```

For now, you can ignore the m_boneID variable - that will come in a future tutorial! The m_location array represents the coordinate of the vertex (X,Y,Z). The two variables store the number of vertices and the actual vertices in a dynamic array which is allocated by the loader.
Next we need to group these vertices into triangles:

```
// Triangle Structure
struct Triangle
{
```

```
float m_vertexNormals[3][3];
float m_s[3], m_t[3];
int m_vertexIndices[3];
};

// Triangles Used
int m_numTriangles;
Triangle *m_pTriangles;
```

Now, the 3 vertices that make up the triangle are stored in m_vertexIndices. These are offsets into the array of m_pVertices. This way each vertex need only be listed once, saving memory (and calculations when it comes to animating later). m_s and m_t are the (s,t) texture coordinates for each of the 3 vertices. The texture used is the one applied to this mesh (which is described next). Finally we have the m_vertexNormals member which stores the normal to each of the 3 vertices. Each normal has 3 float coordinates describing the vector.

The next structure we have in a model is a mesh. A mesh is a group of triangles that all have the same material applied to them. The collection of meshes make up the entire model. The mesh structure is as follows:

```
// Mesh
struct Mesh
{
int m_materialIndex;
int m_numTriangles;
int *m_pTriangleIndices;
};

// Meshes Used
int m_numMeshes;
Mesh *m_pMeshes;
```

This time you have m_pTriangleIndices storing the triangles in the mesh in the same way as the triangle stored indicies to its vertices. It will be dynamically allocated because the number of triangles in a mesh is not known in advance, and is specified by m_numTriangles. Finally, m_materialIndex is the index of the material (texture and lighting coeffecients) to use for the mesh. I'll show you the material structure below:

```
// Material Properties
struct Material
{
float m_ambient[4], m_diffuse[4], m_specular[4], m_emissive[4];
float m_shininess;
GLuint m_texture;
char *m_pTextureFilename;
};

// Materials Used
int m_numMaterials;
Material *m_pMaterials;
```

Here we have all the standard lighting coeffecients in the same format as OpenGL: ambient, diffuse, specular, emissive and shininess. We also have the texture object m_texture and the filename (dynamically allocated) of the texture so that it can be reloaded if the OpenGL context is lost.

The Code - Loading the Model

Now, on to loading the model. You will notice there is a pure virtual function called loadModelData, which takes the filename of the model as an argument. What happens is we create a derived class, MilkshapeModel, which implements this function, filling in the protected data structures mentioned above. Lets look at that function now:

```
bool MilkshapeModel::loadModelData( const char *filename )
{
ifstream inputFile( filename, ios::in | ios::binary | ios::nocreate );
if ( inputFile.fail())
return false; // "Couldn't Open The Model File."
```

First, the file is opened. It is a binary file, hence the ios::binary qualifier. If it is not found, the function returns false to indicate an error.

```
inputFile.seekg( 0, ios::end );
long fileSize = inputFile.tellg();
inputFile.seekg( 0, ios::beg );
```

The above code determines the size of the file in bytes.

```
byte *pBuffer = new byte[fileSize];
inputFile.read( pBuffer, fileSize );
inputFile.close();
```

Then the file is read into a temporary buffer in its entirety.

```
const byte *pPtr = pBuffer;
MS3DHeader *pHeader = ( MS3DHeader* )pPtr;
pPtr += sizeof( MS3DHeader );
```

```
if ( strncmp( pHeader->m_ID, "MS3D000000", 10 ) != 0 )
return false; // "Not A Valid Milkshape3D Model File."

if ( pHeader->m_version < 3 || pHeader->m_version > 4 )
return false; // "Unhandled File Version. Only Milkshape3D Version 1.3 And 1.4 Is Supported."
```

Now, a pointer is acquired to out current position in the file, pPtr. A pointer to the header is saved, and then the pointer is advanced past the header. You will notice several MS3D... structures being used here. These are declared at the top of MilkshapeModel.cpp, and come directly from the file format specification. The fields of the header are checked to make sure that this is a valid file we are reading.

```
int nVertices = *( word* )pPtr;
m_numVertices = nVertices;
m_pVertices = new Vertex[nVertices];
pPtr += sizeof( word );

int i;
for ( i = 0; i < nVertices; i++ )
{
MS3DVertex *pVertex = ( MS3DVertex* )pPtr;
m_pVertices[i].m_boneID = pVertex->m_boneID;
memcpy( m_pVertices[i].m_location, pVertex->m_vertex, sizeof( float )*3 );
pPtr += sizeof( MS3DVertex );
}
```

The above code reads each of the vertex structures in the file. First memory is allocated in the model for the vertices, and then each is parsed from the file as the pointer is advanced. Several calls to memcpy will be used in this function, which copies the contents of the small arrays easily. The m_boneID member can still be ignored for now - its for skeletal animation!

```
int nTriangles = *( word* )pPtr;
m_numTriangles = nTriangles;
m_pTriangles = new Triangle[nTriangles];
pPtr += sizeof( word );

for ( i = 0; i < nTriangles; i++ )
{
MS3DTriangle *pTriangle = ( MS3DTriangle* )pPtr;
int vertexIndices[3] = { pTriangle->m_vertexIndices[0], pTriangle->m_vertexIndices[1],
pTriangle->m_vertexIndices[2] };
float t[3] = { 1.0f-pTriangle->m_t[0], 1.0f-pTriangle->m_t[1], 1.0f-pTriangle->m_t[2] };
memcpy( m_pTriangles[i].m_vertexNormals, pTriangle->m_vertexNormals, sizeof( float )*3*3 );
memcpy( m_pTriangles[i].m_s, pTriangle->m_s, sizeof( float )*3 );
memcpy( m_pTriangles[i].m_t, t, sizeof( float )*3 );
memcpy( m_pTriangles[i].m_vertexIndices, vertexIndices, sizeof( int )*3 );
pPtr += sizeof( MS3DTriangle );
}
```

As for the vertices, this part of the function stores all of the triangles in the model. While most of it involves just copying the arrays from one structure to another, you'll notice the difference for the vertexIndices and t arrays. In the file, the vertex indices are stores as an array of word values, but in the model they are int values for consistency and simplicity (no nasty casting needed). So this just converts the 3 values to integers. The t values are all set to 1.0-(original value). The reason for this is that OpenGL uses a lower-left coordinate system, whereas Milkshape uses an upper-left coordinate system for its texture coordinates. This reverses the y coordinate.

```
int nGroups = *( word* )pPtr;
m_numMeshes = nGroups;
m_pMeshes = new Mesh[nGroups];
pPtr += sizeof( word );
for ( i = 0; i < nGroups; i++ )
{
pPtr += sizeof( byte ); // Flags
pPtr += 32; // Name

word nTriangles = *( word* )pPtr;
pPtr += sizeof( word );
int *pTriangleIndices = new int[nTriangles];
for ( int j = 0; j < nTriangles; j++ )
{
pTriangleIndices[j] = *( word* )pPtr;
pPtr += sizeof( word );
}

char materialIndex = *( char* )pPtr;
pPtr += sizeof( char );

m_pMeshes[i].m_materialIndex = materialIndex;
m_pMeshes[i].m_numTriangles = nTriangles;
m_pMeshes[i].m_pTriangleIndices = pTriangleIndices;
}
```

The above code loads the mesh data structures (also called groups in Milkshape3D). Since the number of triangles varies from mesh to mesh, there is no standard structure to read. Instead, they are taken field by field. The memory for the triangle indices is dynamically allocated within the mesh and read one at a time.

```
int nMaterials = *( word* )pPtr;
m_numMaterials = nMaterials;
m_pMaterials = new Material[nMaterials];
pPtr += sizeof( word );
for ( i = 0; i < nMaterials; i++ )
{
MS3DMaterial *pMaterial = ( MS3DMaterial* )pPtr;
memcpy( m_pMaterials[i].m_ambient, pMaterial->m_ambient, sizeof( float )*4 );
memcpy( m_pMaterials[i].m_diffuse, pMaterial->m_diffuse, sizeof( float )*4 );
memcpy( m_pMaterials[i].m_specular, pMaterial->m_specular, sizeof( float )*4 );
memcpy( m_pMaterials[i].m_emissive, pMaterial->m_emissive, sizeof( float )*4 );
m_pMaterials[i].m_shininess = pMaterial->m_shininess;
m_pMaterials[i].m_pTextureFilename = new char[strlen( pMaterial->m_texture )+1];
strcpy( m_pMaterials[i].m_pTextureFilename, pMaterial->m_texture );
pPtr += sizeof( MS3DMaterial );
}

reloadTextures();
```

Lastly, the material information is taken from the buffer. This is done in the same way as those above, copying each of the lighting coefficients into the new structure. Also, new memory is allocated for the texture filename, and it is copied into there. The final call to reloadTextures is used to actually load the textures and bind them to OpenGL texture objects. That function, from the Model base class, is described later.

```
delete[] pBuffer;

return true;
}
```

The last fragment frees the temporary buffer now that all the data has been copied and returns successfully.

So at this point, the protected member variables of the Model class are filled with the model information. You'll note also that this is the only code in MilkshapeModel because it is the only code specific to Milkshape3D. Now, before the model can be rendered, it is necessary to load the textures for each of its materials. This is done with the following code:

```
void Model::reloadTextures()
{
for ( int i = 0; i < m_numMaterials; i++ )
if ( strlen( m_pMaterials[i].m_pTextureFilename ) > 0 )
m_pMaterials[i].m_texture = LoadGLTexture( m_pMaterials[i].m_pTextureFilename );
else
m_pMaterials[i].m_texture = 0;
}
```

For each material, the texture is loaded using a function from NeHe's base code (slightly modified from it's previous version). If the texture filename was an empty string, then it is not loaded, and instead the texture object identifier is set to 0 to indicate there is no texture.

The Code - Drawing the Model

Now we can start the code to draw the model! This is not difficult at all now that we have a careful arrangement of the data structures in memory.

```
void Model::draw()
{
GLboolean texEnabled = glIsEnabled( GL_TEXTURE_2D );
```

This first part saves the state of texture mapping within OpenGL so that the function does not disturb it. Note however that it does not preserve the material properties in the same way.

Now we loop through each of the meshes and draw them individually:

```
// Draw By Group
for ( int i = 0; i < m_numMeshes; i++ )
{
```

m_pMeshes[i] will be used to reference the current mesh. Now, each mesh has its own material properties, so we set up the OpenGL states according to that. If the materialIndex of the mesh is -1 however, there is no material for this mesh and it is drawn with the OpenGL defaults.

```
int materialIndex = m_pMeshes[i].m_materialIndex;
if ( materialIndex >= 0 )
{
glMaterialfv( GL_FRONT, GL_AMBIENT, m_pMaterials[materialIndex].m_ambient );
glMaterialfv( GL_FRONT, GL_DIFFUSE, m_pMaterials[materialIndex].m_diffuse );
glMaterialfv( GL_FRONT, GL_SPECULAR, m_pMaterials[materialIndex].m_specular );
glMaterialfv( GL_FRONT, GL_EMISSION, m_pMaterials[materialIndex].m_emissive );
glMaterialf( GL_FRONT, GL_SHININESS, m_pMaterials[materialIndex].m_shininess );

if ( m_pMaterials[materialIndex].m_texture > 0 )
{
```

```
glBindTexture( GL_TEXTURE_2D, m_pMaterials[materialIndex].m_texture );
glEnable( GL_TEXTURE_2D );
}
else
glDisable( GL_TEXTURE_2D );
}
else
{
glDisable( GL_TEXTURE_2D );
}
```

The material properties are set according to the values stored in the model. Note that the texture is only bound and enabled if it is greater than 0. If it is set to 0, you'll recall, there was no texture, so texturing is disabled. Texturing is also disabled if there was no material at all for the mesh.

```
glBegin( GL_TRIANGLES );
{
for ( int j = 0; j < m_pMeshes[i].m_numTriangles; j++ )
{
int triangleIndex = m_pMeshes[i].m_pTriangleIndices[j];
const Triangle* pTri = &m_pTriangles[triangleIndex];

for ( int k = 0; k < 3; k++ )
{
int index = pTri->m_vertexIndices[k];

glNormal3fv( pTri->m_vertexNormals[k] );
glTexCoord2f( pTri->m_s[k], pTri->m_t[k] );
glVertex3fv( m_pVertices[index].m_location );
}
}
}
glEnd();
}
```

The above section does the rendering of the triangles for the model. It loops through each of the triangles for the mesh, and then draws each of its three vertices, including the normal and texture coordinates. Remember that each triangle in a mesh and likewise each vertex in a triangle is indexed into the total model arrays (these are the two index variables used). pTri is a pointer to the current triangle in the mesh used to simplify the code following it.

```
if ( texEnabled )
glEnable( GL_TEXTURE_2D );
else
glDisable( GL_TEXTURE_2D );
}
```

This final fragment of code sets the texture mapping state back to its original value.

The only other code of interest in the Model class is the constructor and destructor. These are self explanatory. The constructor initializes all members to 0 (or NULL for pointers), and the destructor deletes the dynamic memory for all of the model structures. You should note that if you call the loadModelData function twice for one Model object, you will get memory leaks. Be careful!

The final topic I will discuss here is the changes to the base code to render using the new Model class, and where I plan to go from here in a future tutorial introducing skeletal animation.

```
Model *pModel = NULL; // Holds The Model Data
```

At the top of the code in Lesson32.cpp the model is declared, but not initialised. It is created in WinMain:

```
pModel = new MilkshapeModel();
if ( pModel->loadModelData( "data/model.ms3d" ) == false )
{
MessageBox( NULL, "Couldn't load the model data/model.ms3d", "Error", MB_OK | MB_ICONERROR );
return 0; // If Model Didn't Load, Quit
}
```

The model is created here, and not in InitGL because InitGL gets called everytime we change the screen mode (losing the OpenGL context). But the model doesn't need to be reloaded, as its data remains intact. What doesn't remain intact are the textures that were bound to texture objects when we loaded the object. So the following line is added to InitGL:

```
pModel->reloadTextures();
```

This takes the place of calling LoadGLTextures as we used to. If there was more than one model in the scene, then this function must be called for all of them. If you get white objects all of a sudden, then your textures have been thrown away and not reloaded correctly.

Finally there is a new DrawGLScene function:

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The View
gluLookAt( 75, 75, 75, 0, 0, 0, 0, 1, 0 );

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

pModel->draw();

yrot+=1.0f;
return TRUE; // Keep Going
}
```

Simple? We clear the colour buffer, set the identity into the model/view matrix, and then set an eye projection with gluLookAt. If you haven't used gluLookAt before, essentially it places the camera at the position of the first 3 parameters, places the center of the scene at the position of the next 3 parameters, and the last 3 parameters describe the vector that is "up". In this case, we look from (75, 75, 75) to (0,0,0) - as the model is drawn about (0,0,0) unless you translate before drawing it - and the positive Y-axis is facing up. The function must be called first, and after loading the identity to behave in this fashion.

To make it a bit more interesting, the scene gradually rotates around the y-axis with glRotatef.

Finally, the model is drawn with its draw member function. It is drawn centered at the origin (assuming it was modelled around the origin in Milkshape 3D!), so If you want to position or rotate or scale it, simply call the appropriate GL functions before drawing it. Voila! To test it out - try making your own models in Milkshape (or use its import function), and load them instead by changing the line in WinMain. Or add them to the scene and draw several models!

What Next?

In a future tutorial for NeHe Productions, I will explain how to extend this class structure to incorporate skeletal animation. And if I get around to it, I will write more loader classes to make the program more versatile.

The step to skeletal animation is not as large as it may seem, although the math involved is much more tricky. If you don't understand much about matrices and vectors, now is the time to read up them! There are several resources on the web that can help you out.

See you then!

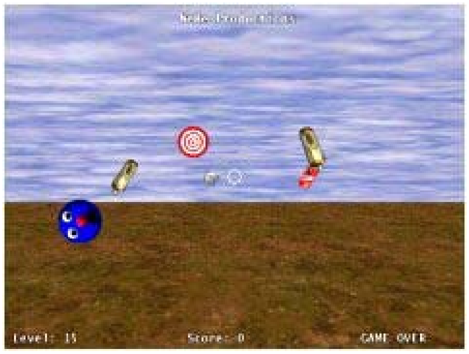
Some information about Brett Porter: Born in Australia, he studied at the University of Wollongong, recently graduating with a BCompSc and a BMath. He began programming in BASIC 12 years ago on a Commodore 64 "clone" called the VZ300, but soon moved up to Pascal, Intel assembly, C++ and Java. During the last few years 3D programming has become an interest and OpenGL has become his graphics API of choice. For more information visit his homepage at: http://rsn.gamedev.net.

A follow up to this tutorial on Skeletal Animation can be found on Brett's homepage. Visit the link above!

Brett Porter

Jeff Molofee (**NeHe**)

Lesson 32
Picking, Alpha Blending, Alpha Testing, Sorting



Welcome to Tutorial 32. This tutorial is probably the largest tutorial I have written to date. Over 1000 lines of Code and more than 1540 lines of HTML. This is also the first tutorial to use my new NeHeGL basecode. This tutorial took a long time to write, but I think it was worth the wait. Some of the topics I cover in this tutorial are: Alpha Blending, Alpha Testing, Reading The Mouse, Using Both Ortho And Perspective At The Same Time, Displaying A Custom Cursor, Manually Sorting Objects By Depth, Animating Frames From A Single Texture and most important, you will learn all about PICKING!

The original version of this tutorial displayed three objects on the screen that would change color when you clicked on them. How exciting is that!?! Not exciting at all! As always, I wanted to impress you guys with a super cool tutorial. I wanted the tutorial to be exciting, packed full of information and of course... nice to look at. So, after weeks of coding, the tutorials is done! Even if you don't code you might enjoy this tutorial. It's a complete game. The object of the game is to shoot as many targets as you can before your morale hits rock bottom or your hand cramps up and you can no longer click the mouse button.

I'm sure there will be critics, but I'm very happy with this tutorial! I've taken dull topics such as picking and sorting object by depth and turned them into something fun!

Some quick notes about the code. I will only discuss the code in lesson33.cpp. There have been a few minor changes in the NeHeGL code. The most important change is that I have added mouse support to WindowProc(). I also added int mouse_x, mouse_y to store mouse movement. In NeHeGL.h the following two lines of code were added: extern int mouse_x; & extern int mouse_y;

The textures used in this tutorial were made in Adobe Photoshop. Each .TGA file is a 32 bit image with an alpha channel. If you are not sure how to add an alpha channel to an image buy yourself a good book, browse the net or read the built in help in Adobe Photoshop. The entire process is very similar to the way I created masks in the masking tutorial. Load your object into Adobe Photoshop (or some other art program that supports the alpha channel). Use select by color range to select the area around your object. Copy that area. Create a new image. Paste the selection into the new image. Negate the image so the area where you image should be is black. Make the area around it white. Select the entire image and copy it. Go back to the original image and create an alpha channel. Paste the black and white mask that you just created into the alpha channel. Save the image as a 32 bit .TGA file. Make sure preserve transparency is checked, and make sure you save it uncompressed!

As always I hope you enjoy the tutorial. I'm interested to hear what you think of it. If you have any questions or you find any mistakes, let me know. I rushed though parts of the tutorial, so if you find any part really hard to understand, send me some email and I'll try to explain things differently or in more detail!

```
#include <windows.h> // Header File For Windows
#include <stdio.h> // Header File For Standard Input / Output
#include <stdarg.h> // Header File For Variable Argument Routines
#include <gl\gl.h> // Header File For The OpenGL32 Library
#include <gl\glu.h> // Header File For The GLu32 Library
#include <time.h> // For Random Seed
#include "NeHeGL.h" // Header File For NeHeGL
```

In lesson 1, I preached about the proper way to link to the OpenGL libraries. In Visual C++ click on project, settings and then the link tab. Move down to object/library modules and add OpenGL32.lib, GLu32.lib and GLaux.lib. Failing to include a required library will cause the compiler to spout out error after error. Something you don't want happening! To make matters worse, if you only include the libaries in debug mode, and someone tries to build your code in release mode... more errors. There are alot of people looking for code. Most of them are new to programming. They grab your code, and try to compile it. They get errors, delete the code and move on.

The code below tells the compiler to link to the required libraries. A little more typing, but alot less headache in the long run. For this tutorial, we will link to the OpenGL32 library, the GLu32 library and the WinMM library (for playing sound). In this tutorial we will be loading .TGA files so we don't need the GLaux library.

```
#pragma comment( lib, "opengl32.lib" ) // Search For OpenGL32.lib While Linking
#pragma comment( lib, "glu32.lib" ) // Search For GLu32.lib While Linking
#pragma comment( lib, "winmm.lib" ) // Search For WinMM Library While Linking
```

The 3 lines below check to see if CDS_FULLSCREEN has been defined by your compiler. If it has not been defined, we manually give CDS_FULLSCREEN a value of 4. For those of you that are completely lost right now... Some compilers do not give CDS_FULLSCREEN a value and will return an error message if CDS_FULLSCREEN is used! To prevent an error message, we check to see if CDS_FULLSCREEN has been defined and if not, we manually define it. Makes life easier for everyone.

We then declare DrawTargets, and set up variables for our window and keyboard handling. If you don't understand declarations, read through the MSDN glossary. Keep in mind, I'm not teaching C/C++, buy a good book if you need help with the NON GL code!

```
#ifndef CDS_FULLSCREEN // CDS_FULLSCREEN Is Not Defined By Some
#define CDS_FULLSCREEN 4 // Compilers. By Defining It This Way,
#endif // We Can Avoid Errors

void DrawTargets(); // Declaration

GL_Window* g_window;
Keys* g_keys;
```

The following section of code sets up our user defined variables. base will be used for our font display lists. roll will be used to move the ground and create the illusion of rolling clouds. level should be pretty straight forward (we start off on level 1). miss keeps track of how many objects were missed. It's also used to show the players morale (no misses means a high morale). kills keeps track of how many targets were hit each level. score will keep a running total of the number of objects hit, and game will be used to signal game over!

The last line lets us pass structures to our compare function. The qsort routine expects the last parameter to be type type (const *void, const *void).

```
// User Defined Variables
GLuint base; // Font Display List
GLfloat roll; // Rolling Clouds
GLint level=1; // Current Level
GLint miss; // Missed Targets
GLint kills; // Level Kill Counter
GLint score; // Current Score
bool game; // Game Over?

typedef int (*compfn)(const void*, const void*); // Typedef For Our Compare Function
```

Now for our objects structure. This structure holds all the information about an object. The direction it's rotating, if it's been hit, it's location on the screen, etc.

A quick rundown of the variables... rot specifies the direction we want to rotate the object. hit will be FALSE if the object has not yet been hit. If the object was hit or manually flagged as being hit, the value of hit will be TRUE.

The variable frame is used to cycle through the frames of animation for our explosion. As frame is increased the explosion texture changes. More on this later in the tutorial.

To keep track of which direction our object is moving, we have a variable called dir. dir can be one of 4 values: 0 - object is moving Left, 1 - object is moving right, 2 - object is moving up and finally 3 - object is moving down.

texid can be any number from 0 to 4. Zero represents the BlueFace texture, 1 is the Bucket texture, 2 is the Target texture , 3 is the Coke can texture and 4 is the Vase texture. Later in the load texture code, you will see that the first 5 textures are the target images.

Both x and y are used to position the object on the screen. x represents where the object is on the x-axis, and y the location of the object on the y-axis.

The objects rotate on the z-axis based on the value of spin. Later in the code, we will increase or decrease spin based on the direction the object is travelling.

Finally, distance keeps track of how far into the screen our object is. distance is an extremely important variable, we will use it to calculate the left and right sides of the screen, and to sort the objects so the objects in the distance are drawn before the objects up close.

```
struct objects {
GLuint rot; // Rotation (0-None, 1-Clockwise, 2-Counter Clockwise)
bool hit; // Object Hit?
GLuint frame; // Current Explosion Frame
GLuint dir; // Object Direction (0-Left, 1-Right, 2-Up, 3-Down)
GLuint texid; // Object Texture ID
GLfloat x; // Object X Position
GLfloat y; // Object Y Position
GLfloat spin; // Object Spin
GLfloat distance; // Object Distance
};
```

No real reason to explain the code below. We are loading TGA images in this tutorial instead of bitmaps. The structure below is used to store image data, as well as information about the TGA image. Read the tutorial on loading TGA files if you need a detailed explanation of the code below.

```
typedef struct // Create A Structure
{
GLubyte *imageData; // Image Data (Up To 32 Bits)
GLuint bpp; // Image Color Depth In Bits Per Pixel.
GLuint width; // Image Width
GLuint height; // Image Height
GLuint texID; // Texture ID Used To Select A Texture
} TextureImage; // Structure Name
```

The following code sets aside room for our 10 textures and 30 objects. If you plan to add more objects to the game make sure you increase the value from 30 to however many objects you want.

```
TextureImage textures[10]; // Storage For 10 Textures

objects object[30]; // Storage For 30 Objects
```

I didn't want to limit the size of each object. I wanted the vase to be taller than the can, I wanted the bucket to be wider than the vase. To make life easy, I create a structure that holds the objects width (w) and height (h).

I then set the width and height of each object in the last line of code. To get the coke cans width, I would check size[3].w. The Blueface is 0, the Bucket is 1, and the Target is 2, etc. The width is represented by w. Make sense?

```
struct dimensions { // Object Dimensions
GLfloat w; // Object Width
GLfloat h; // Object Height
};

// Size Of Each Object: Blueface, Bucket, Target, Coke, Vase
dimensions size[5] = { {1.0f,1.0f}, {1.0f,1.0f}, {1.0f,1.0f}, {0.5f,1.0f}, {0.75f,1.5f} };
```

The following large section of code loads our TGA images and converts them to textures. It's the same code I used in lesson 25 so if you need a detailed description go back and read lesson 25.

I use TGA images because they are capable of having an alpha channel. The alpha channel tells OpenGL which parts of the image are transparent and which parts are opaque. The alpha channel is created in an art program, and is saved inside the .TGA image. OpenGL loads the image, and uses the alpha channel to set the amount of transparency for each pixel in the image.

```
bool LoadTGA(TextureImage *texture, char *filename) // Loads A TGA File Into Memory
{
GLubyte TGAheader[12]={0,0,2,0,0,0,0,0,0,0,0,0}; // Uncompressed TGA Header
GLubyte TGAcompare[12]; // Used To Compare TGA Header
GLubyte header[6]; // First 6 Useful Bytes From The Header
GLuint bytesPerPixel; // Holds Number Of Bytes Per Pixel Used In The TGA File
GLuint imageSize; // Used To Store The Image Size When Setting Aside Ram
GLuint temp; // Temporary Variable
GLuint type=GL_RGBA; // Set The Default GL Mode To RBGA (32 BPP)

FILE *file = fopen(filename, "rb"); // Open The TGA File

if( file==NULL || // Does File Even Exist?
fread(TGAcompare,1,sizeof(TGAcompare),file)!=sizeof(TGAcompare) || // Are There 12 Bytes To
Read?
memcmp(TGAheader,TGAcompare,sizeof(TGAheader))!=0 || // Does The Header Match What We Want?
fread(header,1,sizeof(header),file)!=sizeof(header)) // If So Read Next 6 Header Bytes
{
if (file == NULL) // Does The File Even Exist? *Added Jim Strong*
return FALSE; // Return False
else // Otherwise
{
fclose(file); // If Anything Failed, Close The File
return FALSE; // Return False
}
}

texture->width = header[1] * 256 + header[0]; // Determine The TGA Width (highbyte*256+lowbyte)
texture->height = header[3] * 256 + header[2]; // Determine The TGA Height
(highbyte*256+lowbyte)

if( texture->width <=0 || // Is The Width Less Than Or Equal To Zero
texture->height <=0 || // Is The Height Less Than Or Equal To Zero
(header[4]!=24 && header[4]!=32)) // Is The TGA 24 or 32 Bit?
{
fclose(file); // If Anything Failed, Close The File
return FALSE; // Return False
}

texture->bpp = header[4]; // Grab The TGA's Bits Per Pixel (24 or 32)
bytesPerPixel = texture->bpp/8; // Divide By 8 To Get The Bytes Per Pixel
imageSize = texture->width*texture->height*bytesPerPixel; // Calculate The Memory Required For
The TGA Data

texture->imageData=(GLubyte *)malloc(imageSize); // Reserve Memory To Hold The TGA Data

if( texture->imageData==NULL || // Does The Storage Memory Exist?
fread(texture->imageData, 1, imageSize, file)!=imageSize) // Does The Image Size Match The
Memory Reserved?
{
if(texture->imageData!=NULL) // Was Image Data Loaded
free(texture->imageData); // If So, Release The Image Data

fclose(file); // Close The File
return FALSE; // Return False
}

for(GLuint i=0; i<int(imageSize); i+=bytesPerPixel) // Loop Through The Image Data
{ // Swaps The 1st And 3rd Bytes ('R'ed and 'B'lue)
temp=texture->imageData[i]; // Temporarily Store The Value At Image Data 'i'
texture->imageData[i] = texture->imageData[i + 2]; // Set The 1st Byte To The Value Of The 3rd
Byte
texture->imageData[i + 2] = temp; // Set The 3rd Byte To The Value In 'temp' (1st Byte Value)
}

fclose (file); // Close The File

// Build A Texture From The Data
glGenTextures(1, &texture[0].texID); // Generate OpenGL texture IDs

glBindTexture(GL_TEXTURE_2D, texture[0].texID); // Bind Our Texture
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Linear Filtered
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Linear Filtered

if (texture[0].bpp==24) // Was The TGA 24 Bits
{
type=GL_RGB; // If So Set The 'type' To GL_RGB
}

glTexImage2D(GL_TEXTURE_2D, 0, type, texture[0].width, texture[0].height, 0, type,
GL_UNSIGNED_BYTE, texture[0].imageData);

return true; // Texture Building Went Ok, Return True
}
```

The 2D texture font code is the same code I have used in previous tutorials. However, there are a few small changes. The thing you will notice is that we are only generating 95 display lists. If you look at the font texture, you will see there are only 95 characters counting the space at the top left of the image. The second thing you will notice is we divide by 16.0f for cx and we only divide by 8.0f for cy. The reason we do this is because the font texture is 256 pixels wide, but only half as tall (128 pixels). So to calculate cx we divide by 16.0f and to calculate cy we divide by half that (8.0f).

If you do not understand the code below, go back and read through Lesson 17. The font building code is explained in detail in lesson 17!

```
GLvoid BuildFont(GLvoid) // Build Our Font Display List
{
base=glGenLists(95); // Creating 95 Display Lists
glBindTexture(GL_TEXTURE_2D, textures[9].texID); // Bind Our Font Texture
for (int loop=0; loop<95; loop++) // Loop Through All 95 Lists
{
float cx=float(loop%16)/16.0f; // X Position Of Current Character
float cy=float(loop/16)/8.0f; // Y Position Of Current Character

glNewList(base+loop,GL_COMPILE); // Start Building A List
glBegin(GL_QUADS); // Use A Quad For Each Character
glTexCoord2f(cx, 1.0f-cy-0.120f); glVertex2i(0,0); // Texture / Vertex Coord (Bottom Left)
glTexCoord2f(cx+0.0625f, 1.0f-cy-0.120f); glVertex2i(16,0); // Texutre / Vertex Coord (Bottom
Right)
glTexCoord2f(cx+0.0625f, 1.0f-cy); glVertex2i(16,16); // Texture / Vertex Coord (Top Right)
glTexCoord2f(cx, 1.0f-cy); glVertex2i(0,16); // Texture / Vertex Coord (Top Left)
glEnd(); // Done Building Our Quad (Character)
glTranslated(10,0,0); // Move To The Right Of The Character
glEndList(); // Done Building The Display List
} // Loop Until All 256 Are Built
}
```

The printing code is the code is also from lesson 17, but has been modified to allow us to print the score, level and morale to the screen (variables that continually change).

```
GLvoid glPrint(GLint x, GLint y, const char *string, ...) // Where The Printing Happens
```

```
{
char text[256]; // Holds Our String
va_list ap; // Pointer To List Of Arguments

if (string == NULL) // If There's No Text
return; // Do Nothing

va_start(ap, string); // Parses The String For Variables
vsprintf(text, string, ap); // And Converts Symbols To Actual Numbers
va_end(ap); // Results Are Stored In Text

glBindTexture(GL_TEXTURE_2D, textures[9].texID); // Select Our Font Texture
glPushMatrix(); // Store The Modelview Matrix
glLoadIdentity(); // Reset The Modelview Matrix
glTranslated(x,y,0); // Position The Text (0,0 - Bottom Left)
glListBase(base-32); // Choose The Font Set
glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // Draws The Display List Text
glPopMatrix(); // Restore The Old Projection Matrix
}
```

This code will be called later in the program by qsort. It compares the distance in two structures and return -1 if the first structures distance was less than the seconds structures distance, 1 if the first structures distance is greater than the second structures distance and 0 if the distance is the same in both structures.

```
int Compare(struct objects *elem1, struct objects *elem2) // Compare Function *** MSDN CODE
MODIFIED FOR THIS TUT ***
{
if ( elem1->distance < elem2->distance) // If First Structure distance Is Less Than The Second
return -1; // Return -1
else if (elem1->distance > elem2->distance) // If First Structure distance Is Greater Than The
Second
return 1; // Return 1
else // Otherwise (If The distance Is Equal)
return 0; // Return 0
}
```

The InitObject() code is where we set up each object. We start off by setting rot to 1. This gives the object clockwise rotation. Then we set the explosion animation to frame 0 (we don't want the explosion to start halfway through the animation). Next we set hit to FALSE, meaning the object has not yet been hit or set to self destruct. To select an object texture, texid is assigned a random value from 0 to 4. Zero is the blueface texture and 4 is the vase texture. This gives us one of 5 random objects.

The variable distance will be a random number from -0.0f to -40.0f (4000/100 is 40). When we actually draw the object, we translate another 10 units into the screen. So when the object are drawn, they will be drawn from -10.0f to -50.0f units into the screen (not to close, and not too far). I divide the random number by 100.0f to get a more accurate floating point value.

After assigning a random distance, we then give the object a random y value. We don't want the object any lower than -1.5f, otherwise it will be under the ground, and we dont want the object any higher than 3.0f. So to stay in that range our random number can not be any higher than 4.5f (-1.5f+4.5f=3.0f).

To calculate the x position, we use some tricky math. We take our distance and we subtract 15.0f from it. Then we divide the result by 2 and subtract 5*level. Finally, we subtract a random amount from 0.0f to 5 multiplied by the current level. We subtract the 5*level and the random amount from 0.0f to 5*level so that our object appears further off the screen on higher levels. If we didn't, the objects would appear one after another, making it even more difficult to hit all the targets than it already is.

Finally we choose a random direction (dir) from 0 (left) to 1 (right).

To make things easier to understand in regards to the x position, I'll write out a quick example. Say our distance is -30.0f and the current level is 1: object[num].x=((-30.0f-15.0f)/2.0f)-(5*1)-float(rand()%(5*1));
object[num].x=(-45.0f/2.0f)-5-float(rand()%5);
object[num].x=(-22.5f)-5-{lets say 3.0f};
object[num].x=(-22.5f)-5-{3.0f};
object[num].x=-27.5f-{3.0f};
object[num].x=-30.5f;

Now keeping in mind that we move 10 units into the screen before we draw our objects, and the distance in the example above is -30.0f. it's safe to say our actual distance into the screen will be -40.0f. Using the perspective code in the NeHeGL.cpp file, it's safe to assume that if the distance is -40.0f, the far left edge of the screen will be -20.0f and the far right will be +20.0f. In the code above our x value is -22.5f (which would be JUST off the left side of the screen). We then subtract 5 and our random value of 3 which guarantees the object will start off the screen (at -30.5f) which means the object would have to move roughly 8 units to the right before it even appeared on the screen.

```
GLvoid InitObject(int num) // Initialize An Object
{
object[num].rot=1; // Clockwise Rotation
object[num].frame=0; // Reset The Explosion Frame To Zero
object[num].hit=FALSE; // Reset Object Has Been Hit Status To False
object[num].texid=rand()%5; // Assign A New Texture
object[num].distance=-(float(rand()%4001)/100.0f); // Random Distance
object[num].y=-1.5f+(float(rand()%451)/100.0f); // Random Y Position
// Random Starting X Position Based On Distance Of Object And Random Amount For A Delay
```

```
(Positive Value)
object[num].x=((object[num].distance-15.0f)/2.0f)-(5*level)-float(rand()%(5*level));
object[num].dir=(rand()%2); // Pick A Random Direction
```

Now we check to see which direction the object is going to be travelling. The code below checks to see if the object is moving left. If it is, we have to change the rotation so that the object is spinning counter clockwise. We do this by changing the value of rot to 2.

Our x value by default is going to be a negative number. However, the right side of the screen would be a positive value. So the last thing we do is negate the current x value. In english, we make the x value a positive value instead of a negative value.

```
if (object[num].dir==0) // Is Random Direction Right
{
object[num].rot=2; // Counter Clockwise Rotation
object[num].x=-object[num].x; // Start On The Left Side (Negative Value)
}
```

Now we check the texid to find out what object the computer has randomly picked. If texid is equal to 0, the computer has picked the Blueface object. The blueface guys always roll across the ground. To make sure they start off at ground level, we manually set the y value to -2.0f.

```
if (object[num].texid==0) // Blue Face
object[num].y=-2.0f; // Always Rolling On The Ground
```

Next we check to see if texid is 1. If so, the computer has selected the Bucket. The bucket doesn't travel from left to right, it falls from the sky. The first thing we have to do is set dir to 3. This tells the computer that our bucket is falling or moving down.

Our initial code assumes the object will be travelling from left to right. Because the bucket is falling down, we have to give it a new random x value. If we didn't, the bucket would never be visible. It would fall either far off the left side of the screen or far off the right side of the screen. To assign a new value we randomly choose a value based on the distance into the screen. Instead of subtracting 15, we only subtract 10. This gives us a little less range, and keeps the object ON the screen instead of off the side of the screen. Assuming our distance was -30.0f, we would end up with a random value from 0.0f to 40.0f. If you're asking yourself, why from 0.0f to 40.0f? Shouldn't it be from 0.0f to -40.0f? The answer is easy. The rand() function always returns a positive number. So whatever number we get back will be a positive value. Anyways... back to the story. So we have a positive number from 0.0f to 40.0f. We then add the distance (a negative value) minus 10.0f divided by 2. As an example... assuming the random value returned is say 15 and the distance is -30.0f:

object[num].x=float(rand()%int(-30.0f-10.0f))+((-30.0f-10.0f)/2.0f);

object[num].x=float(rand()%int(-40.0f)+(-40.0f)/2.0f);

object[num].x=float(15 {assuming 15 was returned))+(-20.0f);

object[num].x=15.0f-20.0f;

object[num].x=-5.0f;

The last thing we have to do is set the y value. We want the bucket to drop from the sky. We don't want it falling through the clouds though. So we set the y value to 4.5f. Just a little below the clouds.

```
if (object[num].texid==1) // Bucket
{
object[num].dir=3; // Falling Down
object[num].x=float(rand()%int(object[num].distance-10.0f))+((object[num].distance-10.0f)/2.0f);
object[num].y=4.5f; // Random X, Start At Top Of The Screen
}
```

We want the target to pop out of the ground and up into the air. We check to make sure the object is indeed a target (texid is 2). If so, we set the direction (dir) to 2 (up). We use the exact same code as above to get a random x location.

We don't want the target to start above ground. So we set it's initial y value to -3.0f (under the ground). We then subtract a random value from 0.0f to 5 multiplied by the current level. We do this so that the target doesn't INSTANTLY appear. On higher levels we want a delay before the target appears. Without a delay, the targets would pop out one after another, giving you very little time to hit them.

```
if (object[num].texid==2) // Target
{
object[num].dir=2; // Start Off Flying Up
object[num].x=float(rand()%int(object[num].distance-10.0f))+((object[num].distance-10.0f)/2.0f);
object[num].y=-3.0f-float(rand()%(5*level)); // Random X, Start Under Ground + Random Value
}
```

All of the other objects travel from left to right, so there is no need to assign any values to the remaining objects. They should work just fine with the random values they were assigned.

Now for the fun stuff! *"For the alpha blending technique to work correctly, the transparent primitives must be drawn in back to front* *order and must not intersect"*. When drawing alpha blended objects, it is very important that objects in the distance are drawn first, and objects up close are drawn last.

The reason is simple... The Z buffer prevents OpenGL from drawing pixels that are behind things that have already been drawn. So what ends up happening is objects drawn behind transparent objects do not show up. What you end up seeing is a square shape around overlapping objects... Not pretty!

We already know the depth of each object. So after initializing a new object, we can get around this problem by sorting the objects using the qsort function (quick sort). By sorting the objects, we can be sure that the first object drawn is the object furthest away. That way when we draw the objects, starting at the first object, the objects in the distance will be drawn first. Objects that are closer (drawn later) will see the previously drawn objects behind them, and will blend properly!

As noted in the line comments I found this code in the MSDN after searching the net for hours looking for a solution. It works good and allows you to sort entire structures. qsort takes 4 parameters. The first parameter points to the object array (the array to be sorted). The second parameter is the number of arrays we want to sort... of course we want to sort through all the object currently being displayed (which is level). The third parameter specifies the size of our objects structure and the fourth parameter points to our Compare() function.

There is probably a better way to sort structures, but qsort() works... It's quick, convenient and easy to use!

It's important to note, that if you wanted to use the glAlphaFunc() and glEnable(GL_ALPHA_TEST), sorting is not necessary. However, using the Alpha Function you are restricted to completely transparent or completely opaque blending, there is no in between. Sorting and using the Blendfunc() is a little more work, but it allows for semi-transparent objects.

```
// Sort Objects By Distance: Beginning Address Of Our object Array *** MSDN CODE MODIFIED FOR
THIS TUT ***
// Number Of Elements To Sort
// Size Of Each Element
// Pointer To Our Compare Function
qsort((void *) &object, level, sizeof(struct objects), (compfn)Compare );
}
```

The init code is same as always. The first two lines grab information about our window and our keyboard handler. We then use srand() to create a more random game based on the time. After that we load our TGA images and convert them to textures using LoadTGA(). The first 5 images are objects that will streak across the screen. Explode is our explosion animation, ground and sky make up the background scene, crosshair is the crosshair you see on the screen representing your current mouse location, and finally, the font image is the font used to display the score, title, and morale. If any of the images fail to load FALSE is returned, and the program shuts down. It's important to note that this base code will not return an INIT FAILED error message.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Any OpenGL Initialization Goes Here
{
g_window = window;
g_keys = keys;

srand( (unsigned)time( NULL ) ); // Randomize Things

if ((!LoadTGA(&textures[0],"Data/BlueFace.tga")) || // Load The BlueFace Texture
(!LoadTGA(&textures[1],"Data/Bucket.tga")) || // Load The Bucket Texture
(!LoadTGA(&textures[2],"Data/Target.tga")) || // Load The Target Texture
(!LoadTGA(&textures[3],"Data/Coke.tga")) || // Load The Coke Texture
(!LoadTGA(&textures[4],"Data/Vase.tga")) || // Load The Vase Texture
(!LoadTGA(&textures[5],"Data/Explode.tga")) || // Load The Explosion Texture
(!LoadTGA(&textures[6],"Data/Ground.tga")) || // Load The Ground Texture
(!LoadTGA(&textures[7],"Data/Sky.tga")) || // Load The Sky Texture
(!LoadTGA(&textures[8],"Data/Crosshair.tga")) || // Load The Crosshair Texture
(!LoadTGA(&textures[9],"Data/Font.tga"))) // Load The Crosshair Texture
{
return FALSE; // If Loading Failed, Return False
}
```

If all of the images loaded and were successfully turned into textures, we can continue with initialization. The font texture is loaded, so it's safe to build our font. We do this by jumping to BuildFont().

We then set up OpenGL. The background color is set to black, the alpha is also set to 0.0f. The depth buffer is set up and enabled with less than or equal testing.

The glBlendFunc() is a VERY important line of code. We set the blend function to (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA). This blends the object with whats on the screen using the alpha values stored in the objects texture. After setting the blend mode, we enable blending. We then enable 2D texture mapping, and finally, we enable GL_CULL_FACE. This removes the back face from each object (no point in wasting cycles drawing something we can't see). We draw all of our quads with a counter clockwise winding so the proper face is culled.

Earlier in the tutorial I talked about using the glAlphaFunc() instead of alpha blending. If you want to use the Alpha Function, comment out the 2 lines of blending code and uncomment the 2 lines under glEnable(GL_BLEND). You can also comment out the qsort() function in the InitObject() section of code.

The program should run ok, but the sky texture will not be there. The reason is because the sky texture has an alpha value of 0.5f. When I was talking about the Alpha Function earlier on, I mentioned that it only works with alpha values of 0 or 1. You will have to modify the alpha channel for the sky texture if you want it to appear! Again, if you decide to use the Alpha Function instead, you don't have to sort the objects. Both methods have the good points! Below is a quick quote from the SGI site:

"The alpha function discards fragments instead of drawing them into the frame buffer. Therefore sorting of the primitives is not necessary (unless some other mode like alpha blending is enabled). The disadvantage is that pixels must be completely opaque or completely transparent".

```
BuildFont(); // Build Our Font Display List

glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Black Background
glClearDepth(1.0f); // Depth Buffer Setup
glDepthFunc(GL_LEQUAL); // Type Of Depth Testing
glEnable(GL_DEPTH_TEST); // Enable Depth Testing
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Enable Alpha Blending (disable alpha
testing)
glEnable(GL_BLEND); // Enable Blending (disable alpha testing)
// glAlphaFunc(GL_GREATER,0.1f); // Set Alpha Testing (disable blending)
// glEnable(GL_ALPHA_TEST); // Enable Alpha Testing (disable blending)
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glEnable(GL_CULL_FACE); // Remove Back Face
```

At this point in the program, none of the objects have been defined. So we loop through all thirty objects calling InitObject() for each object.

```
for (int loop=0; loop<30; loop++) // Loop Through 30 Objects
InitObject(loop); // Initialize Each Object

return TRUE; // Return TRUE (Initialization Successful)
}
```

In our init code, we called BuildFont() which builds our 95 display lists. The following line of code deletes all 95 display lists before the program quits.

```
void Deinitialize (void) // Any User DeInitialization Goes Here
{
glDeleteLists(base,95); // Delete All 95 Font Display Lists
}
```

Now for the tricky stuff... The code that does the actual selecting of the objects. The first line of code below allocates a buffer that we can use to store information about our selected objects into. The variable hits will hold the number of hits detected while in selection mode.

```
void Selection(void) // This Is Where Selection Is Done
{
GLuint buffer[512]; // Set Up A Selection Buffer
GLint hits; // The Number Of Objects That We Selected
```

In the code below, we check to see if the game is over (FALSE). If it is, there is no point in selecting anything, so we return (exit). If the game is still active (TRUE), we play a gunshot sound using the Playsound() command. The only time Selection() is called is when the mouse button has been pressed, and every time the button is pressed, we want to play the gunshot sound. The sound is played in async mode so that it doesn't halt the program while the sound is playing.

```
if (game) // Is Game Over?
return; // If So, Don't Bother Checking For Hits

PlaySound("data/shot.wav",NULL,SND_ASYNC); // Play Gun Shot Sound
```

Now we set up a viewport. viewport[] will hold the current x, y, length and width of the current viewport (OpenGL Window).

glGetIntegerv(GL_VIEWPORT, viewport) gets the current viewport boundries and stores them in viewport[]. Initially, the boundries are equal the the OpenGL window dimensions. glSelectBuffer(512, buffer) tells OpenGL to use buffer for it's selection buffer.

```
// The Size Of The Viewport. [0] Is <x>, [1] Is <y>, [2] Is <length>, [3] Is <width>
GLint viewport[4];

// This Sets The Array <viewport> To The Size And Location Of The Screen Relative To The Window
glGetIntegerv(GL_VIEWPORT, viewport);
glSelectBuffer(512, buffer); // Tell OpenGL To Use Our Array For Selection
```

All of the code below is very important. The first line puts OpenGL in selection mode. In selection mode, nothing is drawn to the screen. Instead, information about objects rendered while in selection mode will be stored in the selection buffer.

Next we initialize the name stack by calling glInitNames() and glPushName(0). It's important to note that if the program is not in selection mode, a call to glPushName() will be ignored. Of course we are in selection mode, but it's something to keep in mind.

```
// Puts OpenGL In Selection Mode. Nothing Will Be Drawn. Object ID's and Extents Are Stored In
The Buffer.
(void) glRenderMode(GL_SELECT);

glInitNames(); // Initializes The Name Stack
glPushName(0); // Push 0 (At Least One Entry) Onto The Stack
```

After preparing the name stack, we have to to restrict drawing to the area just under our crosshair. In order to do this we have to select the projection matrix. After selecting the projection matrix we push it onto the stack. We then reset the projection matrix using glLoadIdentity().

We restrict drawing using gluPickMatrix(). The first parameter is our current mouse position on the x-axis, the second parameter is the current mouse position on the y-axis, then the width and height of the picking region. Finally the current viewport[]. The viewport[] indicates the current viewport boundaries. mouse_x and mouse_y will be the center of the picking region.

```
glMatrixMode(GL_PROJECTION); // Selects The Projection Matrix
glPushMatrix(); // Push The Projection Matrix
glLoadIdentity(); // Resets The Matrix

// This Creates A Matrix That Will Zoom Up To A Small Portion Of The Screen, Where The Mouse Is.
gluPickMatrix((GLdouble) mouse_x, (GLdouble) (viewport[3]-mouse_y), 1.0f, 1.0f, viewport);
```

Calling gluPerspective() multiplies the perspective matrix by the pick matrix which restricts the drawing to the area requested by gluPickMatrix().

We then switch to the modelview matrix and draw our targets by calling DrawTargets(). We draw the targets in DrawTargets() and not in Draw() because we only want selection to check for hits with objects (targets) and not the sky, ground or crosshair.

After drawing our targets, we switch back to the projection matrix and pop the stored matrix off the stack. We then switch back to the modelview matrix.

the last line of code below switches back to render mode so that objects we draw actually appear on the screen. hits will hold the number of objects that were rendered in the viewing area requested by gluPickMatrix().

```
// Apply The Perspective Matrix
gluPerspective(45.0f, (GLfloat) (viewport[2]-viewport[0])/(GLfloat) (viewport[3]-viewport[1]),
0.1f, 100.0f);
glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix
DrawTargets(); // Render The Targets To The Selection Buffer
glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
glPopMatrix(); // Pop The Projection Matrix
glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix
hits=glRenderMode(GL_RENDER); // Switch To Render Mode, Find Out How Many
```

Now we check to see if there were more than 0 hits recorded. if so, we set choose to equal the name of the first object drawn into the picking area. depth holds how deep into the screen, the object is.

Each hit takes 4 items in the buffer. The first item is the number of names on the name stack when the hit occured. The second item is the minimum z value of all the verticies that intersected the viewing area at the time of the hit. The third item is the maximum z value of all the vertices that intersected the viewing area at the time of the hit and the last item is the content of the name stack at the time of the hit (name of the object). We are only interested in the minimum z value and the object name in this tutorial.

```
if (hits > 0) // If There Were More Than 0 Hits
{
int choose = buffer[3]; // Make Our Selection The First Object
int depth = buffer[1]; // Store How Far Away It Is
```

We then loop through all of the hits to make sure none of the objects are closer than the first object hit. If we didn't do this, and two objects were overlapping, the first object hit might behind another object, and clicking the mouse would take away the first object, even though it was behind another object. When you shoot at something, the closest object should be the object that gets hit.

So, we check through all of the hits. Remember that each object takes 4 items in the buffer, so to search through each hit we have to multiply the current loop value by 4. We add 1 to get the depth of each object hit. If the depth is less than the the current selected objects depth, we store the name of the closer object in choose and we store the depth of the closer object in depth. After we have looped through all of our hits, choose will hold the name of the closest object hit, and depth will hold the depth of the closest object hit.

```
for (int loop = 1; loop < hits; loop++) // Loop Through All The Detected Hits
{
// If This Object Is Closer To Us Than The One We Have Selected
if (buffer[loop*4+1] < GLuint(depth))
{
choose = buffer[loop*4+3]; // Select The Closer Object
depth = buffer[loop*4+1]; // Store How Far Away It Is
}
}
```

All we have to do is mark the object as being hit. We check to make sure the object has not already been hit. If it has not been hit, we mark it as being hit by setting hit to TRUE. We increase the players score by 1 point, and we increase the kills counter by 1.

```
if (!object[choose].hit) // If The Object Hasn't Already Been Hit
{
object[choose].hit=TRUE; // Mark The Object As Being Hit
score+=1; // Increase Score
```

```
kills+=1; // Increase Level Kills
```

I use kills to keep track of how many objects have been destroyed on each level. I wanted each level to have more objects (making it harder to get through the level). So I check to see if the players kills is greater than the current level multiplied by 5. On level 1, the player only has to kill 5 objects (1*5). On level 2 the player has to kill 10 objects (2*5), progressively getting harder each level.

So, the first line of code checks to see if kills is higher than the level multiplied by 5. If so, we set miss to 0. This sets the player morale back to 10 out of 10 (the morale is 10-miss). We then set kills to 0 (which starts the counting process over again).

Finally, we increase the value of level by 1 and check to see if we've hit the last level. I have set the maximum level to 30 for the following two reasons... Level 30 is insanely difficult. I am pretty sure no one will ever have that good of a game. The second reason... At the top of the code, we only set up 30 objects. If you want more objects, you have to increase the value accordingly.

It is VERY important to note that you can have a maximum of 64 objects on the screen (0-63). If you try to render 65 or more objects, picking becomes confused, and odd things start to happen. Everything from objects randomly exploding to your computer crashing. It's a physical limit in OpenGL (just like the 8 lights limit).

If by some chance you are a god, and you finish level 30, the level will no longer increase, but your score will. Your morale will also reset to 10 every time you finish the 30th level.

```
if (kills>level*5) // New Level Yet?
{
miss=0; // Misses Reset Back To Zero
kills=0; // Reset Level Kills
level+=1; // Increase Level
if (level>30) // Higher Than 30?
level=30; // Set Level To 30 (Are You A God?)
}
}
}
}
```

Update() is where I check for key presses, and update object movement. One of the nice things about Update() is the milliseconds timer. You can use the milliseconds timer to move objects based on the amount of time that has passed since Update() was last called. It's important to note that moving object based on time keeps the objects moving at the same speed on any processor... BUT there are drawbacks! Lets say you have an object moving 5 units in 10 seconds. On a fast system, the computer will move the object half a unit every second. On a slow system, it could be 2 seconds before the update procedure is even called. So when the object moves, it will appear to skip a spot. The animation will not be as smooth on a slower system. (Note: this is just an exaggerated example... computers update ALOT faster than once every two seconds).

Anyways... with that out of the way... on to the code. The code below checks to see if the escape key is being pressed. If it is, we quit the application by calling TerminateApplication(). g_window holds the information about our window.

```
void Update(DWORD milliseconds) // Perform Motion Updates Here
{
if (g_keys->keyDown[VK_ESCAPE]) // Is ESC Being Pressed?
{
TerminateApplication (g_window); // Terminate The Program
}
```

The code below checks to see if the space bar is pressed and the game is over. If both conditions are true, we initialize all 30 object (give them new directions, textures, etc). We set game to FALSE, telling the program the game is no longer over. We set the score back to 0, the level back to 1, the player kills to 0 and finally we set the miss variable back to zero. This restarts the game on the first level with full morale and a score of 0.

```
if (g_keys->keyDown[' '] && game) // Space Bar Being Pressed After Game Has Ended?
{
for (int loop=0; loop<30; loop++) // Loop Through 30 Objects
InitObject(loop); // Initialize Each Object

game=FALSE; // Set game (Game Over) To False
score=0; // Set score To 0
level=1; // Set level Back To 1
kills=0; // Zero Player Kills
miss=0; // Set miss (Missed Shots) To 0
}
```

The code below checks to see if the F1 key has been pressed. If F1 is being pressed, ToggleFullscreen will switch from windowed to fullscreen mode or fullscreen mode to windowed mode.

```
if (g_keys->keyDown[VK_F1]) // Is F1 Being Pressed?
{
ToggleFullscreen (g_window); // Toggle Fullscreen Mode
}
```

To create the illusion of rolling clouds and moving ground, we decrease roll by .00005f multiplied by the number of milliseconds that have passed. This keeps the clouds moving at the same speed on all systems (fast or slow).

We then set up a loop to loop through all of the objects on the screen. Level 1 has one object, level 10 has 10 objects, etc.

```
roll-=milliseconds*0.00005f; // Roll The Clouds

for (int loop=0; loop<level; loop++) // Loop Through The Objects
{
```

We need to find out which way the object should be spinning. We do this by checking the value of rot. If rot equals 1, we need to spin the object clockwise. To do this, we decrease the value of spin. We decrease spin by 0.2f multiplied by value of loop plus the number of milliseconds that have passed. By using milliseconds the objects will rotate the same speed on all systems. Adding loop makes each NEW object spin a little faster than the last object. So object 2 will spin faster than object 1 and object 3 will spin faster than object 2.

```
if (object[loop].rot==1) // If Rotation Is Clockwise
object[loop].spin-=0.2f*(float(loop+milliseconds)); // Spin Clockwise
```

Next we check to see if rot equals 2. If rot equals 2, we need to spin counter clockwise. The only difference from the code above is that we are increasing the value of spin instead of decreasing it. This causes the object to spin in the opposite direction.

```
if (object[loop].rot==2) // If Rotation Is Counter Clockwise
object[loop].spin+=0.2f*(float(loop+milliseconds)); // Spin Counter Clockwise
```

Now for the movement code. We check the value of dir if it's equal to 1, we increase the objects x value based on the milliseconds passed multiplied by 0.012f. This moves the object right. Because we use milliseconds the objects should move the same speed on all systems.

```
if (object[loop].dir==1) // If Direction Is Right
object[loop].x+=0.012f*float(milliseconds); // Move Right
```

If dir equals 0, the object is moving left. We move the object left by decreasing the objects x value. Again we decrease x based on the amount of time that has passed in milliseconds multiplied by our fixed value of 0.012f.

```
if (object[loop].dir==0) // If Direction Is Left
object[loop].x-=0.012f*float(milliseconds); // Move Left
```

Only two more directions to watch for. This time we check to see if dir equals 2. If so, we increase the objects y value. This causes the object to move UP the screen. Keep in mind the positive y axis is at the top of the screen and the negative y axis is at the bottom. So increasing y moves from the bottom to the top. Again movement is based on time passed.

```
if (object[loop].dir==2) // If Direction Is Up
object[loop].y+=0.012f*float(milliseconds); // Move Up
```

The last direction our object can travel is down. If dir equals three, we want to move the object down the screen. We do this by increasing the objects y value based on the amount of time that has passed. Notice we move down slower than we move up. When an object is falling, our fixed falling rate is 0.0025f. When we move up, the fixed rate is 0.012f.

```
if (object[loop].dir==3) // If Direction Is Down
object[loop].y-=0.0025f*float(milliseconds); // Move Down
```

After moving our objects we have to check if they are still in view. The code below first checks to see where our object is on the screen. We can roughly calculate how far left an object can travel by taking the objects distance into the screen minus 15.0f (to make sure it's a little past the screen) and dividing it by 2. For those of you that don't already know... If you are 20 units into the screen, depending on the way you set up the perspective, you have roughly 10 units from the left of the screen to the center and 10 from the center to the right. so -20.0f(distance)-15.0f(extra padding)=-35.0f... divide that by 2 and you get -17.5f. That's roughly 7.5 units off the left side of the screen. Meaning our object is completely out of view.

Anyways... after making sure the object is far off the left side of the screen, we check to see if it was moving left (dir=0). If it's not moving left, we don't care if it's off the left side of the screen!

Finally, we check to see if the object was hit. If the object is off the left of the screen, it's travelling left and it wasn't hit, it's too late for the player to hit it. So we increase the value of miss. This lowers morale and increases the number of missed targets. We set the objects hit value to TRUE so the computer thinks it's been hit. This forces the object to self destruct (allowing us to give the object a new texture, directions, spin, etc).

```
// If We Are To Far Left, Direction Is Left And The Object Was Not Hit
if ((object[loop].x<(object[loop].distance-15.0f)/2.0f) && (object[loop].dir==0) &&
!object[loop].hit)
{
miss+=1; // Increase miss (Missed Object)
object[loop].hit=TRUE; // Set hit To True To Manually Blow Up The Object
}
```

The following code does the exact same thing as the code above, but instead of checking to see if we've gone off the left side of the screen, we check to see if it's gone off the right side of the screen. We also check to make sure the object is moving right and not some other direction. If the object is off the screen, we increase the value of miss and self destruct the object by telling our program it's been hit.

```
// If We Are To Far Right, Direction Is Left And The Object Was Not Hit
if ((object[loop].x>-(object[loop].distance-15.0f)/2.0f) && (object[loop].dir==1) &&
!object[loop].hit)
{
miss+=1; // Increase miss (Missed Object)
object[loop].hit=TRUE; // Set hit To True To Manually Blow Up The Object
}
```

The falling code is pretty straight forward. We check to see if the object has just about hit the ground. We don't want it to fall through the ground which is at -3.0f. Instead, we check to see if the object is below -2.0f. We then check to make sure the object is indeed falling (dir=3) and that the object has not yet been hit. If the object is below -2.0f on the y axis, we increase miss and set the objects hit variable to TRUE (causing it to self destruct as it hits the ground... nice effect).

```
// If We Are To Far Down, Direction Is Down And The Object Was Not Hit
if ((object[loop].y<-2.0f) && (object[loop].dir==3) && !object[loop].hit)
{
miss+=1; // Increase miss (Missed Object)
object[loop].hit=TRUE; // Set hit To True To Manually Blow Up The Object
}
```

Unlike the previous code, the going up code is a little different. We don't want the object to go through the clouds! We check to see if the objects y variable is greater than 4.5f (close to the clouds). We also make sure the object is travelling up (dir=2). If the objects y value is greater than 4.5f, instead of destroying the object, we change it's direction. That way the object will quickly pop out of the ground (remember, it goes up faster than it comes down) and once it gets to high we change its direction so it starts to fall toward the ground.

There is no need to destroy the object, or increase the miss variable. If you miss the object as it's flying into the sky, there's always a chance to hit it as it falls. The falling code will handle the final destruction of the object.

```
if ((object[loop].y>4.5f) && (object[loop].dir==2)) // If We Are To Far Up And The Direction Is
Up
object[loop].dir=3; // Change The Direction To Down
}
}
```

Next we have the object drawing code. I wanted a quick and easy way to draw the game objects, along with the crosshair with as little code as possible. Object takes 3 parameters. First we have the width. The width controls how wide the object will be when it's drawn. Then we have the height. The height controls how tall the object will be when it's drawn. Finally, we have the texid. The texid selects the texture we want to use. If we wanted to draw a bucket, which is texture 1, we would pass a value of 1 for the texid. Pretty simple!

A quick breakdown. We select the texture, and then draw a quad. We use standard texture coordinates so the entire textue is mapped to the face of the quad. The quad is drawn in a counter-clockwise direction (required for culling to work).

```
void Object(float width,float height,GLuint texid) // Draw Object Using Requested Width, Height
And Texture
{
glBindTexture(GL_TEXTURE_2D, textures[texid].texID); // Select The Correct Texture
glBegin(GL_QUADS); // Start Drawing A Quad
glTexCoord2f(0.0f,0.0f); glVertex3f(-width,-height,0.0f); // Bottom Left
glTexCoord2f(1.0f,0.0f); glVertex3f( width,-height,0.0f); // Bottom Right
glTexCoord2f(1.0f,1.0f); glVertex3f( width, height,0.0f); // Top Right
glTexCoord2f(0.0f,1.0f); glVertex3f(-width, height,0.0f); // Top Left
glEnd(); // Done Drawing Quad
}
```

The explosion code takes one parameter. num is the object identifier. In order to create the explosion we need to grab a portion of the explosion texture similar to the way we grab each letter from the font texture. The two lines below calculate the column (ex) and row (ey) from a single number (frame).

The first line below grabs the current frame and divides it by 4. The division by 4 is to slow down the animation. %4 keeps the value in the 0-3 range. If the value is higher than 3 it would wrap around and become 0. If the value is 5 it would become 1. A value of 9 would be 0,1,2,3,0,1,2,3,0. We divide the final result by 4.0f because texture coordinates are in the 0.0f to 1.0f range. Our explosion texture has 4 explosion images from left to right and 4 up and down.

Hopefully you're not completely confused. So if our number before division can only be 0,1,2 or 3 our number after we divide it by 4.0f can only be 0.0f, 0.25f (1/4), 0.50f (2/4) or 0.75f (3/4). This gives us our left to right texture coordinate (ex).

Next we calculate the row (ey). We grab the current object frame and divide it by 4 to slow the animation down a little. We then divide by 4 again to eliminate an entire row. Finally we divide by 4 one last time to get our vertical texture coordinate.

A quick example. If our current frame was 16. ey=((16/4)/4)/4 or 4/4/4 or 0.25f. One row down. If our current frame was 60. ey=((60/4)/4)/4 or 15/4/4 or 3/4 or 0.75f. The reason 15/4 isn't 3.75 is because we are working with integers up until we do the final division. With that in mind, the value of ey can only be one of 4 values... 0.0f, 0.25f, 0.50f or 0.75f. Assuming we stay inside our texture (prevent frame from going over a value of 63).

Hope that made sense... it's simple, but intimidating math.

```
void Explosion(int num) // Draws An Animated Explosion For Object "num"
{
float ex = (float)((object[num].frame/4)%4)/4.0f; // Calculate Explosion X Frame (0.0f - 0.75f)
float ey = (float)((object[num].frame/4)/4)/4.0f; // Calculate Explosion Y Frame (0.0f - 0.75f)
```

Now that we have the texture coordinates, all that's left to do is draw our textured quad. The vertex coordinates are fixed at -1.0f and 1.0f. You will notice we subract ey from 1.0f. If we didn't, the animation would be drawn in the reverse order... The explosion would get bigger, rather than fade out. The effect wont look right!

We bind the explosion texture before we draw the textured quad. Again, the quad is drawn counter-clockwise.

```
glBindTexture(GL_TEXTURE_2D, textures[5].texID); // Select The Explosion Texture
glBegin(GL_QUADS); // Begin Drawing A Quad
glTexCoord2f(ex ,1.0f-(ey )); glVertex3f(-1.0f,-1.0f,0.0f); // Bottom Left
glTexCoord2f(ex+0.25f,1.0f-(ey )); glVertex3f( 1.0f,-1.0f,0.0f); // Bottom Right
glTexCoord2f(ex+0.25f,1.0f-(ey+0.25f)); glVertex3f( 1.0f, 1.0f,0.0f); // Top Right
glTexCoord2f(ex ,1.0f-(ey+0.25f)); glVertex3f(-1.0f, 1.0f,0.0f); // Top Left
glEnd(); // Done Drawing Quad
```

As I mentioned above, the value of frame should not exceed 63 otherwise the animation will start over again. So we increase the value of frame and then we check to see if the value is greater than 63. If it is, we call InitObject(num) which destroys the object and gives it new values to create an entirely new object.

```
object[num].frame+=1; // Increase Current Explosion Frame
if (object[num].frame>63) // Have We Gone Through All 16 Frames?
{
InitObject(num); // Init The Object (Assign New Values)
}
}
```

This section of code draws all of the targets (objects) to the screen. We start off by resetting the modelview matrix. We then translate 10 units into the screen and set up a loop from 0 to the players current level.

```
void DrawTargets(void) // Draws The Targets (Needs To Be Seperate)
{
glLoadIdentity(); // Reset The Modelview Matrix
glTranslatef(0.0f,0.0f,-10.0f); // Move Into The Screen 20 Units
for (int loop=0; loop<level; loop++) // Loop Through 9 Objects
{
```

The first line of code is the secret to picking individual objects. What it does is assigns a name (number) to each object. The first object drawn will be 0. The second object will be 1, etc... If the loop was to hit 29, the last object drawn would be given the name 29. After assigning a name to the object, we push the modelview matrix onto the stack. It's important to note the calls to glLoadName() are ignored if the program is not in selection mode.

We then move to the location on the screen where we want our object to be drawn. We use object[loop].x to position on the x-axis, object[loop].y to position on the y-axis and object[loop].distance to position the object on the z-axis (depth into the screen). We have already translated 10 units into the screen, so the actual distance at which the object will be drawn is going to be object[loop].distance-10.0f.

```
glLoadName(loop); // Assign Object A Name (ID)
glPushMatrix(); // Push The Modelview Matrix
glTranslatef(object[loop].x,object[loop].y,object[loop].distance); // Position The Object (x,y)
```

Before we draw the object, we have to check if it's been hit or not. We do this by checking to see if object[loop].hit is TRUE. if it is, we jump to Explosion(loop) which will draw the explosion animation instead of the actual object. If the object was not hit, we spin the object on it's z-axis by object[loop].spin degrees before we call Object().

Object takes 3 parameters. The first one is the width, the second one is the height and the third one is the number of the texture to use. To get the width and height, we use the array size[object[loop].texid].w and size[object[loop].texid].h. This look up the width and height from our predefined object size array at the beginning of this program. The reason we use object[loop].texid is because it represents the type of object we are drawing. A texid of 0 is always the blueface... a texid of 3 is always the coke can, etc.

After drawing an object, we pop the matrix resetting the view, so our next object is drawn at the proper location on the screen.

```
if (object[loop].hit) // If Object Has Been Hit
{
Explosion(loop); // Draw An Explosion
}
else // Otherwise
{
glRotatef(object[loop].spin,0.0f,0.0f,1.0f); // Rotate The Object
Object(size[object[loop].texid].w,size[object[loop].texid].h,object[loop].texid); // Draw The
Object
}
glPopMatrix(); // Pop The Modelview Matrix
}
}
```

This is where the drawing occurs. We start off by clearing the screen, and resetting our modelview matrix.

```
void Draw(void) // Draw Our Scene
{
```

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
glLoadIdentity(); // Reset The Modelview Matrix
```

Next we push the modelview matrix onto the stack and select the sky texture (texture 7). The sky is made up of 4 textured quads. The first 4 vertices draw the sky way in the distance from the ground straight up. The texture on this quad will roll fairly slowly. The next 4 vertices draw the sky again at the exact same location but the sky texture will roll faster. The two textures will blend together in alpha blending mode to create a neat multilayered effect.

```
glPushMatrix(); // Push The Modelview Matrix
glBindTexture(GL_TEXTURE_2D, textures[7].texID); // Select The Sky Texture
glBegin(GL_QUADS); // Begin Drawing Quads
glTexCoord2f(1.0f,roll/1.5f+1.0f); glVertex3f( 28.0f,+7.0f,-50.0f); // Top Right
glTexCoord2f(0.0f,roll/1.5f+1.0f); glVertex3f(-28.0f,+7.0f,-50.0f); // Top Left
glTexCoord2f(0.0f,roll/1.5f+0.0f); glVertex3f(-28.0f,-3.0f,-50.0f); // Bottom Left
glTexCoord2f(1.0f,roll/1.5f+0.0f); glVertex3f( 28.0f,-3.0f,-50.0f); // Bottom Right

glTexCoord2f(1.5f,roll+1.0f); glVertex3f( 28.0f,+7.0f,-50.0f); // Top Right
glTexCoord2f(0.5f,roll+1.0f); glVertex3f(-28.0f,+7.0f,-50.0f); // Top Left
glTexCoord2f(0.5f,roll+0.0f); glVertex3f(-28.0f,-3.0f,-50.0f); // Bottom Left
glTexCoord2f(1.5f,roll+0.0f); glVertex3f( 28.0f,-3.0f,-50.0f); // Bottom Right
```

To give the illusion that the sky is coming towards the viewer, we draw two more quads, but this time we draw them from way in the distance coming toward the viewer. The first 4 verticies draw slow rolling clouds and the remaining 4 draw faster moving clouds. The two layers will blend together in alpha blending mode to create a multilayered effect. The second layer of clouds is offset by 0.5f so that the two textures don't line up. Same with the two layers of clouds above. The second layer is offset by 0.5f.

The final effect of all 4 quads is a sky that appears to move up way out in the distance and then toward the viewer up high. I could have used a textured half sphere for the sky, but I was too lazy, and the effect is still pretty good as is.

```
glTexCoord2f(1.0f,roll/1.5f+1.0f); glVertex3f( 28.0f,+7.0f,0.0f); // Top Right
glTexCoord2f(0.0f,roll/1.5f+1.0f); glVertex3f(-28.0f,+7.0f,0.0f); // Top Left
glTexCoord2f(0.0f,roll/1.5f+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f); // Bottom Left
glTexCoord2f(1.0f,roll/1.5f+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f); // Bottom Right

glTexCoord2f(1.5f,roll+1.0f); glVertex3f( 28.0f,+7.0f,0.0f); // Top Right
glTexCoord2f(0.5f,roll+1.0f); glVertex3f(-28.0f,+7.0f,0.0f); // Top Left
glTexCoord2f(0.5f,roll+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f); // Bottom Left
glTexCoord2f(1.5f,roll+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f); // Bottom Right
glEnd(); // Done Drawing Quads
```

With the sky out of the way, it's time to draw the ground. We draw the ground starting where the sky texture is the lowest coming towards the viewer. The ground texture rolls at the same speed as the fast moving clouds.

The texture is repeated 7 times from left to right and 4 times from back to front to add a little more detail and to prevent the texture from getting all blocky looking. This is done by increasing the texture coordinates from 0.0f - 1.0f to 0.0f - 7.0f (left to right) and 0.0f - 4.0f (up and down).

```
glBindTexture(GL_TEXTURE_2D, textures[6].texID); // Select The Ground Texture
glBegin(GL_QUADS); // Draw A Quad
glTexCoord2f(7.0f,4.0f-roll); glVertex3f( 27.0f,-3.0f,-50.0f); // Top Right
glTexCoord2f(0.0f,4.0f-roll); glVertex3f(-27.0f,-3.0f,-50.0f); // Top Left
glTexCoord2f(0.0f,0.0f-roll); glVertex3f(-27.0f,-3.0f,0.0f); // Bottom Left
glTexCoord2f(7.0f,0.0f-roll); glVertex3f( 27.0f,-3.0f,0.0f); // Bottom Right
glEnd(); // Done Drawing Quad
```

After drawing the sky and the ground, we jump to the section of code that draws all of our targets (objects) called none other than DrawTargets().

After drawing out targets, we pop the modelview matrix off the stack (restoring it to it's previous state).

```
DrawTargets(); // Draw Our Targets
glPopMatrix(); // Pop The Modelview Matrix
```

The code below draws the crosshair. We start off by grabbing our current window dimensions. We do this in case the window was resized in windowed mode. GetClientRect grabs the dimensions and stores them in window. We then select our projection matrix and push it onto the stack. We reset the view with glLoadIdentity() and then set the screen up in ortho mode instead of perspective. The window will go from 0 to window.right from left to right, and from 0 to window.bottom from the bottom to the top of the screen.

The third parameter of glOrtho() is supposed to be the bottom value, instead I swapped the bottom and top values. I did this so that the crosshair would be rendered in a counter clockwise direction. With 0 at the top and window.bottom at the bottom, the winding would go the opposite direction and the crosshair and text would not appear.

After setting up the ortho view, we select the modelview matrix, and position the crosshair. Because the screen is upside down, we have to invert the mouse as well. Otherwise our crosshair would move down if we moved the mouse up and up if we moved the mouse down. To do this we subtract the current mouse_y value from the bottom of the window (window.bottom).

After translating to the current mouse position, we draw the crosshair. We do this by calling Object(). Instead of units, we are going to specify the width and height in pixels. The crosshair will be 16x16 pixels wide and tall and the texture used to draw the object is texture 8 (the crosshair texture).

I decided to use a custom cursor for two reasons... first and most important, it looks cool, and it can be modified using any art program that supports the alpha channel. Secondly, some video cards do not display a cursor in fullscreen mode. Playing the game without a cursor in fullscreen mode is not easy :)

```
// Crosshair (In Ortho View)
RECT window; // Storage For Window Dimensions
GetClientRect (g_window->hWnd,&window); // Get Window Dimensions
glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
glPushMatrix(); // Store The Projection Matrix
glLoadIdentity(); // Reset The Projection Matrix
glOrtho(0,window.right,0,window.bottom,-1,1); // Set Up An Ortho Screen
glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix
glTranslated(mouse_x,window.bottom-mouse_y,0.0f); // Move To The Current Mouse Position
Object(16,16,8); // Draw The Crosshair
```

This section of code put the title at the top of the screen, and displays the level and score in the bottom left and right corners of the screen. The reason I put this code here is because it's easier to position the text accurately in ortho mode.

```
// Game Stats / Title
glPrint(240,450,"NeHe Productions"); // Print Title
glPrint(10,10,"Level: %i",level); // Print Level
glPrint(250,10,"Score: %i",score); // Print Score
```

This section checks to see if the player has missed more than 10 objects. If so, we set the number of misses (miss) to 9 and we set game to TRUE. Setting the game to TRUE means the game is over!

```
if (miss>9) // Have We Missed 10 Objects?
{
miss=9; // Limit Misses To 10
game=TRUE; // Game Over TRUE
}
```

In the code below, we check to see if game is TRUE. If game is TRUE, we print the GAME OVER messages. If game is false, we print the players morale (out of 10). The morale is calculated by subtracting the players misses (miss) from 10. The more the player misses, the lower his morale.

```
if (game) // Is Game Over?
glPrint(490,10,"GAME OVER"); // Game Over Message
else
glPrint(490,10,"Morale: %i/10",10-miss); // Print Morale #/10
```

The last thing we do is select the projection matrix, restore (pop) our matrix back to it's previous state, set the matrix mode to modelview and flush the buffer to make sure all objects have been rendered.

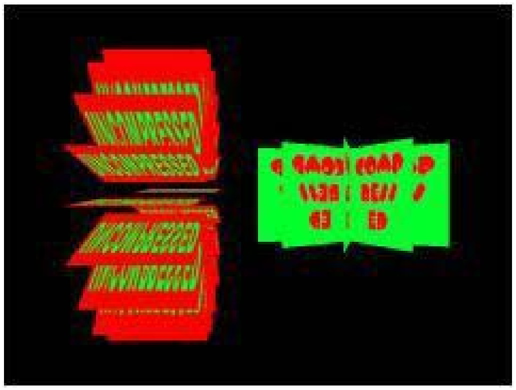
```
glMatrixMode(GL_PROJECTION); // Select The Projection Matrix
glPopMatrix(); // Restore The Old Projection Matrix
glMatrixMode(GL_MODELVIEW); // Select The Modelview Matrix

glFlush(); // Flush The GL Rendering Pipeline
}
```

This tutorial is the result of many late nights, and many many hours of coding & writing HTML. By the end of this tutorial you should have a good understanding of how picking, sorting, alpha blending and alpha testing works. Picking allows you to create interactive point and click software. Everything from games, to fancy GUI's. The best feature of picking is that you don't have to keep track of where your objects are. You assign a name and check for hits. It's that easy! With alpha blending and alpha testing you can make your objects completely solid, or full of holes. The results are great, and you don't have to worry about objects showing through your textures, unless you want them to! As always, I hope you have enjoyed this tutorial, and hope to see some cool games, or projects based on code from this tutorial. If you have any questions or you find mistakes in the tutorial please let me know... I'm only human :)

I could have spent alot more time adding things like physics, more graphics, more sound, etc. This is just a tutorial though! I didn't write it to impress you with bells and whistles. I wrote it to teach you OpenGL with as little confusion as possible. I hope to see some cool modifications to the code. If you add something cool the the tutorial send me the demo. If it's a cool modification I'll post it to the downloads page. If I get enough modifications I may set up a page dedicated to modified versions of this tutorial! I am here to give you a starting point. The rest is up to you :)

Jeff Molofee (**NeHe**)

Lesson 33
Loading Compressed And Uncompressed TGA's



Loading Uncompressed and RunLength Encoded TGA images By Evan "terminate" Pipho.

I've seen lots of people ask around in #gamdev, the gamedev forums, and other places about TGA loading. The following code and explanation will show you how to load both uncompressed TGA files and RLE compressed files. This particular tutorial is geared toward OpenGL, but I plan to make it more universal in the future.

We will begin with the two header files. The first file will hold our texture structure, the second, structures and variables used by the loading code.

Like every header file we need some inclusion guards to prevent the file from being included multiple times.

At the top of the file add these lines:

```
#ifndef __TEXTURE_H__ // See If The Header Has Been Defined Yet
#define __TEXTURE_H__ // If Not, Define It.
```

Then scroll all the way down to the bottom and add:

```
#endif // __TEXTURE_H__ End Inclusion Guard
```

These three lines prevent the file from being included more than once into a file. The rest of the code in the file will be between the first two, and the last line.

Into this header file we we will insert the standard headers we will need for everything we do. Add the following lines after the #define __TGA_H__ command.

```
#pragma comment(lib, "OpenGL32.lib") // Link To Opengl32.lib
#include <windows.h> // Standard Windows header
#include <stdio.h> // Standard Header For File I/O
#include <gl\gl.h> // Standard Header For OpenGL
```

The first header is the standard windows header, the second is for the file I/O functions we will be using later, and the 3rd is the standard OpenGL header file for OpenGL32.lib.

We will need a place to store image data and specifications for generating a texture usable by OpenGL. We will use the following structure.

```
typedef struct
{
GLubyte* imageData; // Hold All The Color Values For The Image.
GLuint bpp; // Hold The Number Of Bits Per Pixel.
GLuint width; // The Width Of The Entire Image.
GLuint height; // The Height Of The Entire Image.
GLuint texID; // Texture ID For Use With glBindTexture.
GLuint type; // Data Stored In * ImageData (GL_RGB Or GL_RGBA)
} Texture;
```

Now for the other, longer head file. Again we will need some includsion guards, same as the last one.

Next come two more structures used during processing of the TGA file.

```
typedef struct
{
GLubyte Header[12]; // File Header To Determine File Type
} TGAHeader;

typedef struct
{
GLubyte header[6]; // Holds The First 6 Useful Bytes Of The File
GLuint bytesPerPixel; // Number Of BYTES Per Pixel (3 Or 4)
GLuint imageSize; // Amount Of Memory Needed To Hold The Image
GLuint type; // The Type Of Image, GL_RGB Or GL_RGBA
GLuint Height; // Height Of Image
GLuint Width; // Width Of Image
GLuint Bpp; // Number Of BITS Per Pixel (24 Or 32)
} TGA;
```

Now we declare some instances of our two structures so we can use them within our code.

```
TGAHeader tgaheader; // Used To Store Our File Header
TGA tga; // Used To Store File Information
```

We need to define a couple file headers so we can tell the program what kinds of file headers are on valid images. The first 12 bytes will be 0 0 2 0 0 0 0 0 0 0 0 0 if it is uncompressed TGA image and 0 0 10 0 0 0 0 0 0 0 0 0 if it an RLE compressed one. These two values allow us to check to see if the file we are reading is valid.

```
// Uncompressed TGA Header
GLubyte uTGAcompare[12] = {0,0, 2,0,0,0,0,0,0,0,0,0};
// Compressed TGA Header
GLubyte cTGAcompare[12] = {0,0,10,0,0,0,0,0,0,0,0,0};
```

Finally we declare a pair of functions to use in the loading process.

```
// Load An Uncompressed File
bool LoadUncompressedTGA(Texture *, char *, FILE *);
// Load A Compressed File
bool LoadCompressedTGA(Texture *, char *, FILE *);
```

Now, on to the cpp file, and the real brunt of the code. I will leave out some of the error message code and the like to make the tutorial shorter and more readable. You may look in the included file (link at the bottom of the article)

Right off the bat we have to include the file we just made so at the top of the file put.

```
#include "tga.h" // Include The Header We Just Made
```

We don't have to include any other files, because we included them inside our header we just completed.

The next thing we see is the first function, which is called LoadTGA(...).

```
// Load A TGA File!
bool LoadTGA(Texture * texture, char * filename)
{
```

It takes two parameters. The first is a pointer to a Texture structure, which you must have declared in your code some where (see included example). The second parameter is a string that tells the computer where to find your texture file.

The first two lines of the function declare a file pointer and then open the file specified by "filename" which was passed to the function in the second parem for reading.

```
FILE * fTGA; // Declare File Pointer
fTGA = fopen(filename, "rb"); // Open File For Reading
```

The next few lines check to make sure that the file opened correctly.

```
if(fTGA == NULL) // If Here Was An Error
{
...Error code...
return false; // Return False
}
```

Next we try to read the first twelve bytes of the file and store them in out TGAHeader structure so we can check on the file type. If fread fails, the file is closed, an error displayed, and the function returns false.

```
// Attempt To Read The File Header
if(fread(&tgaheader, sizeof(TGAHeader), 1, fTGA) == 0)
{
...Error code here...
return false; // Return False If It Fails
}
```

Next we try to determine what type of file it is by comparing our newly aquired header with our two hard coded ones. This will tell us if its compressed, uncompressed, or even if its the wrong file type. For this purpose we will use the memcmp(...) function.

```
// If The File Header Matches The Uncompressed Header
if(memcmp(uTGAcompare, &tgaheader, sizeof(tgaheader)) == 0)
{
// Load An Uncompressed TGA
LoadUncompressedTGA(texture, filename, fTGA);
}
// If The File Header Matches The Compressed Header
else if(memcmp(cTGAcompare, &tgaheader, sizeof(tgaheader)) == 0)
{
// Load A Compressed TGA
LoadCompressedTGA(texture, filename, fTGA);
}
else // If It Doesn't Match Either One
{
...Error code here...
return false; // Return False
}
```

We will begin this section with the loading of an UNCOMPRESSED file. This function is heavily based on NeHe's code in lesson 25.

First thing we come to, as always, is the function header.

```
// Load An Uncompressed TGA!
bool LoadUncompressedTGA(Texture * texture, char * filename, FILE * fTGA)
{
```

This function takes three parameters. The first two are the same as from LoadTGA, and are simply passed on. The third is the file pointer from the previous function so that we dont lose our place.

Next we try to read the next 6 bytes of the file, and store them in tga.header. If it fails, we run some error code, and return false.

```
// Attempt To Read Next 6 Bytes
if(fread(tga.header, sizeof(tga.header), 1, fTGA) == 0)
{
...Error code here...
return false; // Return False
}
```

Now we have all the information we need to calculate the height, width and bpp of our image. We store it in both the texture and local structure.

```
texture->width = tga.header[1] * 256 + tga.header[0]; // Calculate Height
texture->height = tga.header[3] * 256 + tga.header[2]; // Calculate The Width
texture->bpp = tga.header[4]; // Calculate Bits Per Pixel
tga.Width = texture->width; // Copy Width Into Local Structure
tga.Height = texture->height; // Copy Height Into Local Structure
tga.Bpp = texture->bpp; // Copy Bpp Into Local Structure
```

Now we check to make sure that the height and width are at least one pixel, and that the bpp is either 24 or 32. If any of the values are outside their boundries we again display an error, close the file, and leave the function.

```
// Make Sure All Information Is Valid
if((texture->width <= 0) || (texture->height <= 0) || ((texture->bpp != 24) && (texture->bpp
!=32)))
{
...Error code here...
return false; // Return False
}
```

Next we set the type of the image. 24 bit images are type GL_RGB and 32 bit images are type GL_RGBA

```
if(texture->bpp == 24) // Is It A 24bpp Image?
{
texture->type = GL_RGB; // If So, Set Type To GL_RGB
}
else // If It's Not 24, It Must Be 32
{
texture->type = GL_RGBA; // So Set The Type To GL_RGBA
}
```

Now we caclulate the BYTES per pixel and the total size of the image data.

```
tga.bytesPerPixel = (tga.Bpp / 8); // Calculate The BYTES Per Pixel
// Calculate Memory Needed To Store Image
tga.imageSize = (tga.bytesPerPixel * tga.Width * tga.Height);
```

We need some place to store all that image data so we will use malloc to allocate the right amount of memory.

Then we check to make sure memory was allocated, and is not NULL. If there was an error, run error handling code.

```
// Allocate Memory
texture->imageData = (GLubyte *)malloc(tga.imageSize);
if(texture->imageData == NULL) // Make Sure It Was Allocated Ok
{
...Error code here...
return false; // If Not, Return False
}
```

Here we try to read all the image data. If we can't, we trigger error code again.

```
// Attempt To Read All The Image Data
if(fread(texture->imageData, 1, tga.imageSize, fTGA) != tga.imageSize)
{
...Error code here...
return false; // If We Cant, Return False
}
```

TGA files store their image in reverse order than what OpenGL wants so we much change the format from BGR to RGB. To do this we swap the first and third bytes in every pixel.

Steve Thomas Adds: I've got a little speedup in TGA loading code. It concerns switching BGR into RGB using only 3 binary operations. Instead of using a temp variable you XOR the two bytes 3 times.

Then we close the file, and exit the function successfully.

```
// Start The Loop
for(GLuint cswap = 0; cswap < (int)tga.imageSize; cswap += tga.bytesPerPixel)
{
// 1st Byte XOR 3rd Byte XOR 1st Byte XOR 3rd Byte
texture->imageData[cswap] ^= texture->imageData[cswap+2] ^=
texture->imageData[cswap] ^= texture->imageData[cswap+2];
}

fclose(fTGA); // Close The File
return true; // Return Success
}
```

Thats all there is to loading an uncompressed TGA file. Loading a RLE compressed one is only slightly harder. We read the header and collect height/width/bpp as usual, sme as the uncompressed version, so i will just post the code, you can look in the previous pages for a complete explanation.

```
bool LoadCompressedTGA(Texture * texture, char * filename, FILE * fTGA)
{
if(fread(tga.header, sizeof(tga.header), 1, fTGA) == 0)
{
...Error code here...
}
texture->width = tga.header[1] * 256 + tga.header[0];
texture->height = tga.header[3] * 256 + tga.header[2];
texture->bpp = tga.header[4];
tga.Width = texture->width;
tga.Height = texture->height;
tga.Bpp = texture->bpp;
if((texture->width <= 0) || (texture->height <= 0) || ((texture->bpp != 24) && (texture->bpp
!=32)))
{
...Error code here...
} }
tga.bytesPerPixel = (tga.Bpp / 8);
tga.imageSize = (tga.bytesPerPixel * tga.Width * tga.Height);
```

Now we need to allocate the amount of storage for the image to use AFTER we uncompress it, we will use malloc. If memory fails to be allocated, run error code, and return false.

```
// Allocate Memory To Store Image Data
texture->imageData = (GLubyte *)malloc(tga.imageSize);
if(texture->imageData == NULL) // If Memory Can Not Be Allocated...
{
...Error code here...
return false; // Return False
}
```

Next we need to determine how many pixels make up the image. We will store it in the variable "pixelcount"

We also need to store which pixel we are currently on, and what byte of the imageData we are writing to, to avoid overflows and overwriting old data.

We will allocate enough memory to store one pixel.

```
GLuint pixelcount = tga.Height * tga.Width; // Number Of Pixels In The Image
GLuint currentpixel = 0; // Current Pixel We Are Reading From Data
GLuint currentbyte = 0; // Current Byte We Are Writing Into Imagedata
// Storage For 1 Pixel
GLubyte * colorbuffer = (GLubyte *)malloc(tga.bytesPerPixel);
```

Next we have a big loop.

Lets break it down into more manageable chunks.

First we declare a variable in order to store the chunk header. A chunk header dictates wheather the following section is RLE, or RAW, and how long it is. If the one byte header is less than or equal to 127, then it is a RAW header. The value of the header is the number of colors, minus one, that we read ahead and copy into memory, before we hit another header byte. So we add one to the value we get, and then read that many pixels and copy them into the ImageData, just like we did with the uncompressed ones. If the header is ABOVE 127, then it is the number of times that the next pixel value is repeated consequtively. To get the actual number of repetitions we take the value returned and subtract 127 to get rid of the one bit header identifier. Then we read the next one pixel and copy it the said number of times consecutively into the memory.

On to the code. First we read the one byte header.

```
do // Start Loop
{
GLubyte chunkheader = 0; // Variable To Store The Value Of The Id Chunk
if(fread(&chunkheader, sizeof(GLubyte), 1, fTGA) == 0) // Attempt To Read The Chunk's Header
{
...Error code...
return false; // If It Fails, Return False
}
```

Next we will check to see if it a RAW header. If it is, we need to add one to the value to get the total number of pixels following the header.

```
if(chunkheader < 128) // If The Chunk Is A 'RAW' Chunk
{
chunkheader++; // Add 1 To The Value To Get Total Number Of Raw Pixels
```

We then start another loop to read all the color information. It will loop the amout of times specified in the chunk header, and will read and store one pixel each loop.

First we read and verify the pixel data. The data for one pixel will be stored in the colorbuffer variable. Next we will check to see if it a RAW header. If it is, we need to add one to the value to get the total number of pixels following the header.

```
// Start Pixel Reading Loop
for(short counter = 0; counter < chunkheader; counter++)
{
// Try To Read 1 Pixel
if(fread(colorbuffer, 1, tga.bytesPerPixel, fTGA) != tga.bytesPerPixel)
{
...Error code...
return false; // If It Fails, Return False
}
```

The next part in our loop will take the color values stored in colorbuffer and writing them to the imageData varable to be used later. In the process it will flip the data from BGR format to RGB or from BGRA to RGBA depending on the number of bits per pixel. When we are done we increment the current byte, and current pixel counters.

```
texture->imageData[currentbyte] = colorbuffer[2]; // Write The 'R' Byte
texture->imageData[currentbyte + 1 ] = colorbuffer[1]; // Write The 'G' Byte
texture->imageData[currentbyte + 2 ] = colorbuffer[0]; // Write The 'B' Byte
if(tga.bytesPerPixel == 4) // If It's A 32bpp Image...
{
texture->imageData[currentbyte + 3] = colorbuffer[3]; // Write The 'A' Byte
}
// Increment The Byte Counter By The Number Of Bytes In A Pixel
currentbyte += tga.bytesPerPixel;
currentpixel++; // Increment The Number Of Pixels By 1
```

The next section deals with the chunk headers that represent the RLE sections. First thing we do is subtract 127 from the chunkheader to get the amount of times the next color is repeated.

```
else // If It's An RLE Header
{
chunkheader -= 127; // Subtract 127 To Get Rid Of The ID Bit
```

The we attempt to read the next color value.

```
// Read The Next Pixel
if(fread(colorbuffer, 1, tga.bytesPerPixel, fTGA) != tga.bytesPerPixel)
{
...Error code...
return false; // If It Fails, Return False
}
```

Next we begin a loop to copy the pixel we just read into memory multiple times, as dictated by the value from the RLE header.

Then we copy the the color values into the image data, preforming the R and B value switch.

Then we increment the current bytes, and current pixel, so we are in the right spot when we write the values again.

```
// Start The Loop
for(short counter = 0; counter < chunkheader; counter++)
{
// Copy The 'R' Byte
texture->imageData[currentbyte] = colorbuffer[2];
// Copy The 'G' Byte
texture->imageData[currentbyte + 1 ] = colorbuffer[1];
// Copy The 'B' Byte
texture->imageData[currentbyte + 2 ] = colorbuffer[0];
if(tga.bytesPerPixel == 4) // If It's A 32bpp Image
{
// Copy The 'A' Byte
texture->imageData[currentbyte + 3] = colorbuffer[3];
}
currentbyte += tga.bytesPerPixel; // Increment The Byte Counter
currentpixel++; // Increment The Pixel Counter
```

Then we contiune the main loop, as long as we still have pixels left to read.

Last of all we close up the file and return success.

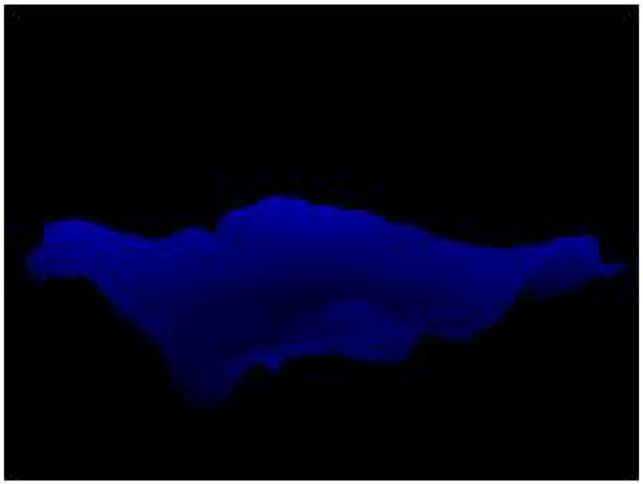
```
while(currentpixel < pixelcount); // More Pixels To Read? ... Start Loop Over
fclose(fTGA); // Close File
return true; // Return Success
}
```

Now you have some image data ready for glGenTextures and glBindTexture. I suggest you check out NeHe's tutorial #6 and #24 for info on these commands. That concludes my first ever tutorial. I do not guarantee my code is error free, though i made an effort to see that it was. Special thanks to Jeff "NeHe" Molofee for his great tutorials and to Trent "ShiningKnight" Polack for helping me revise this tutorial. If you find errors, have suggestions, or comments please feel free to email me (terminate@gdnmail.net), or ICQ me at UIN# 38601160. Enjoy!

Evan Pipho (**Terminate**)

Jeff Molofee (**NeHe**)

Lesson 34
Beautiful Landscapes By Means Of Height Mapping



Welcome to another exciting tutorial! The code for this tutorial was written by Ben Humphrey, and is based on the GL framework from lesson 1. By now you should be a GL expert {grin}, and moving the code into your own base code should be a snap!

This tutorial will teach you how to create cool looking terrain from a height map. For those of you that have no idea what a height map is, I will attempt a crude explanation. A height map is simply... displacement from a surface. For those of you that are still scratching your heads asking yourself "what the heck is this guy talking about!?!"... In english, our heightmap represents low and height points for our landscape. It's completely up to you to decide which shades represent low points and which shades represent high points. It's also important to note that height maps do not have to be images... you can create a height map from just about any type of data. For instance, you could use an audio stream to create a visual height map representation. If you're still confused... keep reading... it will all start to make sense as you go through the tutorial :)

```
#include <windows.h> // Header File For Windows
#include <stdio.h> // Header file For Standard Input/Output ( NEW )
#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

#pragma comment(lib, "opengl32.lib") // Link OpenGL32.lib
#pragma comment(lib, "glu32.lib") // Link Glu32.lib
```

We start off by defining a few important variables. MAP_SIZE is the dimension of our map. In this tutorial, the map is 1024x1024. The STEP_SIZE is the size of each quad we use to draw the landscape. By reducing the step size, the landscape becomes smoother. It's important to note that the smaller the step size, the more of a performance hit your program will take, especially when using large height maps. The HEIGHT_RATIO is used to scale the landscape on the y-axis. A low HEIGHT_RATIO produces flatter mountains. A high HEIGHT_RATIO produces taller / more defined mountains.

Further down in the code you will notice bRender. If bRender is set to true (which it is by default), we will drawn solid polygons. If bRender is set to false, we will draw the landscape in wire frame.

```
#define MAP_SIZE 1024 // Size Of Our .RAW Height Map ( NEW )
#define STEP_SIZE 16 // Width And Height Of Each Quad ( NEW )
#define HEIGHT_RATIO 1.5f // Ratio That The Y Is Scaled According To The X And Z ( 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 TRUE By Default
bool bRender = TRUE; // Polygon Flag Set To TRUE By Default ( NEW )
```

Here we make an array (g_HeightMap[ ]) of bytes to hold our height map data. Since we are reading in a .RAW file that just stores values from 0 to 255, we can use the values as height values, with 255 being the highest point, and 0 being the lowest point. We also create a variable called scaleValue for scaling the entire scene. This gives the user the ability to zoom in and out.

```
BYTE g_HeightMap[MAP_SIZE*MAP_SIZE]; // Holds The Height Map Data ( NEW )

float scaleValue = 0.15f; // Scale Value For The Terrain ( NEW )

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

The ReSizeGLScene() code is the same as lesson 1 except the farthest distance has been changed from 100.0f to 500.0f.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Resize And Initialize The GL Window
{
... CUT ...
}
```

The following code loads in the .RAW file. Not too complex! We open the file in Read/Binary mode. We then check to make sure the file was found and that it could be opened. If there was a problem opening the file for whatever reason, an error message will be displayed.

```
// Loads The .RAW File And Stores It In pHeightMap
void LoadRawFile(LPSTR strName, int nSize, BYTE *pHeightMap)
{
FILE *pFile = NULL;

// Open The File In Read / Binary Mode.
pFile = fopen( strName, "rb" );

// Check To See If We Found The File And Could Open It
if ( pFile == NULL )
{
// Display Error Message And Stop The Function
MessageBox(NULL, "Can't Find The Height Map!", "Error", MB_OK);
return;
}
```

If we've gotten this far, then it's safe to assume there were no problems opening the file. With the file open, we can now read in the data. We do this with fread(). pHeightMap is the storage location for the data (pointer to our g_Heightmap array). 1 is the number of items to load (1 byte at a time), nSize is the maximum number of items to read (the image size in bytes - width of image * height of image). Finally, pFile is a pointer to our file structure!

After reading in the data, we check to see if there were any errors. We store the results in result and then check result. If an error did occur, we pop up an error message.

The last thing we do is close the file with fclose(pFile).

```
// Here We Load The .RAW File Into Our pHeightMap Data Array
// We Are Only Reading In '1', And The Size Is (Width * Height)
fread( pHeightMap, 1, nSize, pFile );

// After We Read The Data, It's A Good Idea To Check If Everything Read Fine
int result = ferror( pFile );

// Check If We Received An Error
if (result)
{
MessageBox(NULL, "Failed To Get Data!", "Error", MB_OK);
}

// Close The File
fclose(pFile);
}
```

The init code is pretty basic. We set the background clear color to black, set up depth testing, polygon smoothing, etc. After doing all that, we load in our .RAW file. To do this, we pass the filename ("Data/Terrain.raw"), the dimensions of the .RAW file (MAP_SIZE * MAP_SIZE) and finally our HeightMap array (g_HeightMap) to LoadRawFile(). This will jump to the .RAW loading code above. The .RAW file will be loaded, and the data will be stored in our Heightmap array (g_HeightMap).

```
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 Calculations

// Here we read read in the height map from the .raw file and put it in our
// g_HeightMap array. We also pass in the size of the .raw file (1024).
LoadRawFile("Data/Terrain.raw", MAP_SIZE * MAP_SIZE, g_HeightMap); // ( NEW )

return TRUE; // Initialization Went OK
}
```

This is used to index into our height map array. When ever we are dealing with arrays, we want to make sure that we don't go outside of them. To make sure that doesn't happen we use %. % will prevent our x / y values from exceeding MAX_SIZE - 1.

We check to make sure pHeightMap points to valid data, if not, we return 0.

Otherwise, we return the value stored at x, y in our height map. By now, you should know that we have to multiply y by the width of the image MAP_SIZE to move through the data. More on this below!

```
int Height(BYTE *pHeightMap, int X, int Y) // This Returns The Height From A Height Map Index
{
int x = X % MAP_SIZE; // Error Check Our x Value
int y = Y % MAP_SIZE; // Error Check Our y Value

if(!pHeightMap) return 0; // Make Sure Our Data Is Valid
```

We need to treat the single array like a 2D array. We can use the equation: index = (x + (y * arrayWidth)). This is assuming we are visualizing it like: pHeightMap[x][y], otherwise it's the opposite: (y + (x * arrayWidth)).

Now that we have the correct index, we will return the height at that index (data at x, y in our array).

```
return pHeightMap[x + (y * MAP_SIZE)]; // Index Into Our Height Array And Return The Height
}
```

Here we set the color for a vertex based on the height index. To make it darker, I start with -0.15f. We also get a ratio of the color from 0.0f to 1.0f by dividing the height by 256.0f. If there is no data this function returns without setting the color. If everything goes ok, we set the color to a shade of blue using glColor3f(0.0f, fColor, 0.0f). Try moving fColor to the red or green spots to change the color of the landscape.

```
void SetVertexColor(BYTE *pHeightMap, int x, int y) // This Sets The Color Value For A
Particular Index
{ // Depending On The Height Index
if(!pHeightMap) return; // Make Sure Our Height Data Is Valid

float fColor = -0.15f + (Height(pHeightMap, x, y ) / 256.0f);

// Assign This Blue Shade To The Current Vertex
glColor3f(0.0f, 0.0f, fColor );
}
```

This is the code that actually draws our landscape. X and Y will be used to loop through the height map data. x, y and z will be used to render the quads making up the landscape.

As always, we check to see if the height map (pHeightMap) contains data. If not, we return without doing anything.

```
void RenderHeightMap(BYTE pHeightMap[]) // This Renders The Height Map As Quads
{
int X = 0, Y = 0; // Create Some Variables To Walk The Array With.
int x, y, z; // Create Some Variables For Readability

if(!pHeightMap) return; // Make Sure Our Height Data Is Valid
```

Since we can switch between lines and quads, we check our render state with the code below. If bRender = True, then we want to render polygons, otherwise we render lines.

```
if(bRender) // What We Want To Render
glBegin( GL_QUADS ); // Render Polygons
else
glBegin( GL_LINES ); // Render Lines Instead
```

Next we actually need to draw the terrain from the height map. To do that, we just walk the array of height data and pluck out some heights to plot our points. If we could see this happening, it would draw the columns first (Y), then draw the rows. Notice that we have a STEP_SIZE. This determines how defined our height map is. The higher the STEP_SIZE, the more blocky the terrain looks, while the lower it gets, the more rounded (smooth) it becomes. If we set STEP_SIZE = 1 it would create a vertex for every pixel in the height map. I chose 16 as a decent size. Anything too much less gets to be insane and slow. Of course, you can increase the number when you get lighting in. Then vertex lighting would cover up the blocky shape. Instead of lighting, we just put a color value associated with every poly to simplify the tutorial. The higher the polygon, the brighter the color is.

```
for ( X = 0; X < MAP_SIZE; X += STEP_SIZE )
for ( Y = 0; Y < MAP_SIZE; Y += STEP_SIZE )
{
// Get The (X, Y, Z) Value For The Bottom Left Vertex
x = X;
y = Height(pHeightMap, X, Y );
z = Y;

// Set The Color Value Of The Current Vertex
SetVertexColor(pHeightMap, x, z);

glVertex3i(x, y, z); // Send This Vertex To OpenGL To Be Rendered

// Get The (X, Y, Z) Value For The Top Left Vertex
x = X;
y = Height(pHeightMap, X, Y + STEP_SIZE );
z = Y + STEP_SIZE ;

// Set The Color Value Of The Current Vertex
SetVertexColor(pHeightMap, x, z);

glVertex3i(x, y, z); // Send This Vertex To OpenGL To Be Rendered

// Get The (X, Y, Z) Value For The Top Right Vertex
x = X + STEP_SIZE;
y = Height(pHeightMap, X + STEP_SIZE, Y + STEP_SIZE );
z = Y + STEP_SIZE ;

// Set The Color Value Of The Current Vertex
SetVertexColor(pHeightMap, x, z);

glVertex3i(x, y, z); // Send This Vertex To OpenGL To Be Rendered

// Get The (X, Y, Z) Value For The Bottom Right Vertex
x = X + STEP_SIZE;
y = Height(pHeightMap, X + STEP_SIZE, Y );
z = Y;

// Set The Color Value Of The Current Vertex
SetVertexColor(pHeightMap, x, z);

glVertex3i(x, y, z); // Send This Vertex To OpenGL To Be Rendered
}
glEnd();
```

After we are done, we set the color back to bright white with an alpha value of 1.0f. If there were other objects on the screen, we wouldn't want them showing up BLUE :)

```
glColor4f(1.0f, 1.0f, 1.0f, 1.0f); // Reset The Color
}
```

For those of you who haven't used gluLookAt(), what it does is position your camera position, your view, and your up vector. Here we set the camera in a obscure position to get a good outside view of the terrain. In order to avoid using such high numbers, we would divide the terrain's vertices by a scale constant, like we do in glScalef() below.

The values of gluLookAt() are as follows: The first three numbers represent where the camera is positioned. So the first three values move the camera 212 units on the x-axis, 60 units on the y-axis and 194 units on the z-axis from our center point. The next 3 values represent where we want the camera to look. In this tutorial, you will notice while running the demo that we are looking a little to the left. We are also look down towards the landscape. 186 is to the left of 212 which gives us the look to the left, and 55 is lower than 60, which gives us the appearance that we are higher than the landscape looking at it with a slight tilt (seeing a bit of the top of it). The value of 171 is how far away from the camera the object is. The last three values tell OpenGL which direction represents up. Our mountains travel upwards on the y-axis, so we set the value on the y-axis to 1. The other two values are set at 0.

gluLookAt can be very intimidating when you first use it. After reading the rough explanation above you may still be confused. My best advise is to play around with the values. Change the camera position. If you were to change the y position of the camera to say 120, you would see more of the top of the landscape, because you would be looking all the way down to 55.

I'm not sure if this will help, but I'm going to break into one of my highly flamed real life "example" explanations :) Lets say you are 6 feet and a bit tall. Lets also assume your eyes are at the 6 foot mark (your eyes represent the camera - 6 foot is 6 units on the y-axis). Now if you were standing in front of a wall that was only 2 feet tall (2 units on the y-axis), you would be looking DOWN at the wall and would be able to see the top of the wall. If the wall was 8 feet tall, you would be looking UP at the wall and you would NOT see the top of the wall. The view would change depending on if you were looking up or down (if you were higher than or lower than the object you are looking at). Hope that makes a bit of sense!

```
int DrawGLScene(GLvoid) // Here's Where We Do All The Drawing
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
glLoadIdentity(); // Reset The Matrix

// Position View Up Vector
gluLookAt(212, 60, 194, 186, 55, 171, 0, 1, 0); // This Determines The Camera's Position And
View
```

This will scale down our terrain so it's a bit easier to view and not so big. We can change this scaleValue by using the UP and DOWN arrows on the keyboard. You will notice that we mupltiply the Y scaleValue by a HEIGHT_RATIO as well. This is so the terrain appears higher and gives it more definition.

```
glScalef(scaleValue, scaleValue * HEIGHT_RATIO, scaleValue);
```

If we pass the g_HeightMap data into our RenderHeightMap() function it will render the terrain in Quads. If you are going to make any use of this function, it might be a good idea to put in an (X, Y) parameter to draw it at, or just use OpenGL's matrix operations (glTranslatef() glRotate(), etc) to position the land exactly where you want it.

```
RenderHeightMap(g_HeightMap); // Render The Height Map

return TRUE; // Keep Going
}
```

The KillGLWindow() code is the same as lesson 1.

```
GLvoid KillGLWindow(GLvoid) // Properly Kill The Window
{
}
```

The CreateGLWindow() code is also the same as lesson 1.

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

The only change in WndProc() is the addition of WM_LBUTTONDOWN. What it does is checks to see if the left mouse button was pressed. If it was, the rendering state is toggled from polygon mode to line mode, or from line mode to polygon mode.

```
LRESULT CALLBACK WndProc( HWND hWnd, // Handle For This Window
UINT uMsg, // Message For This Window
WPARAM wParam, // Additional Message Information
LPARAM lParam) // Additional Message Information
{
switch (uMsg) // Check For Windows Messages
{
case WM_ACTIVATE: // Watch For Window Activate Message
{
if (!HIWORD(wParam)) // Check Minimization State
{
active=TRUE; // Program Is Active
}
else
{
active=FALSE; // Program Is No Longer Active
}

return 0; // Return To The Message Loop
}

case WM_SYSCOMMAND: // Intercept System Commands
{
switch (wParam) // Check System Calls
{
case SC_SCREENSAVE: // Screensaver Trying To Start?
case SC_MONITORPOWER: // Monitor Trying To Enter Powersave?
return 0; // Prevent From Happening
}
break; // Exit
}

case WM_CLOSE: // Did We Receive A Close Message?
{
PostQuitMessage(0); // Send A Quit Message
return 0; // Jump Back
}

case WM_LBUTTONDOWN: // Did We Receive A Left Mouse Click?
{
bRender = !bRender; // Change Rendering State Between Fill/Wire Frame
return 0; // Jump Back
}

case WM_KEYDOWN: // Is A Key Being Held Down?
{
keys[wParam] = TRUE; // If So, Mark It As TRUE
return 0; // Jump Back
}

case WM_KEYUP: // Has A Key Been Released?
{
keys[wParam] = FALSE; // If So, Mark It As FALSE
return 0; // Jump Back
}

case WM_SIZE: // Resize The OpenGL Window
{
ReSizeGLScene(LOWORD(lParam),HIWORD(lParam)); // LoWord=Width, HiWord=Height
return 0; // Jump Back
}
}
// Pass All Unhandled Messages To DefWindowProc
return DefWindowProc(hWnd,uMsg,wParam,lParam);
}
```

No major changes in this section of code. The only notable change is the title of the window. Everything else is the same up until we check for key presses.

```
int WINAPI WinMain( HINSTANCE hInstance, // Instance
HINSTANCE hPrevInstance, // Previous Instance
LPSTR lpCmdLine, // Command Line Parameters
int nCmdShow) // Window Show State
{
MSG msg; // Windows Message Structure
BOOL done=FALSE; // Bool Variable To Exit Loop

// Ask The User Which Screen Mode They Prefer
if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
fullscreen=FALSE; // Windowed Mode
}

// Create Our OpenGL Window
if (!CreateGLWindow("NeHe & Ben Humphrey's Height Map Tutorial", 640, 480, 16, fullscreen))
{
return 0; // Quit If Window Was Not Created
}

while(!done) // Loop That Runs While done=FALSE
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // Is There A Message Waiting?
{
if (msg.message==WM_QUIT) // Have We Received A Quit Message?
{
done=TRUE; // If So done=TRUE
}
else // If Not, Deal With Window Messages
{
TranslateMessage(&msg); // Translate The Message
DispatchMessage(&msg); // Dispatch The Message
}
}
else // If There Are No Messages
{
// Draw The Scene. Watch For ESC Key And Quit Messages From DrawGLScene()
if ((active && !DrawGLScene()) || keys[VK_ESCAPE]) // Active? Was There A Quit Received?
{
done=TRUE; // ESC or DrawGLScene Signalled A Quit
}
else if (active) // Not Time To Quit, Update Screen
{
SwapBuffers(hDC); // Swap Buffers (Double Buffering)
}

if (keys[VK_F1]) // Is F1 Being Pressed?
{
keys[VK_F1]=FALSE; // If So Make Key FALSE
KillGLWindow(); // Kill Our Current Window
fullscreen=!fullscreen; // Toggle Fullscreen / Windowed Mode
// Recreate Our OpenGL Window
if (!CreateGLWindow("NeHe & Ben Humphrey's Height Map Tutorial", 640, 480, 16, fullscreen))
{
return 0; // Quit If Window Was Not Created
}
}
```

The code below lets you increase and decrease the scaleValue. By pressing the up key, the scaleValue is increased, making the landscape larger. By pressing the down key, the scaleValue is decreased making the landscape smaller.

```
if (keys[VK_UP]) // Is The UP ARROW Being Pressed?
scaleValue += 0.001f; // Increase The Scale Value To Zoom In

if (keys[VK_DOWN]) // Is The DOWN ARROW Being Pressed?
scaleValue -= 0.001f; // Decrease The Scale Value To Zoom Out
}
}

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

That's all there is to creating a beautiful height mapped landscape. I hope you appreciate Ben's work! As always, if you find mistakes in the tutorial or the code, please email me, and I will attempt to correct the problem / revise the tutorial.

Once you understand how the code works, play around a little. One thing you could try doing is adding a little ball that rolls across the surface. You already know the height of each section of the landscape, so adding the ball should be no problem. Other things to try: Create the heightmap manually, make it a scrolling landscape, add colors to the landscape to represent snowy peaks / water / etc, add textures, use a plasma effect to create a constantly changing landscape. The possibilities are endless :)

Hope you enjoyed the tut! You can visit Ben's site at: http://www.GameTutorials.com.

Ben Humphrey (**DigiBen**)

Jeff Molofee (**NeHe**)

Lesson 35
Playing AVI Files In OpenGL

I would like to start off by saying that I am very proud of this tutorial. When I first got the idea to code an AVI player in OpenGL thanks to Jonathan de Blok, I had no idea how to open an AVI let alone code an AVI player. I started off by flipping through my collection of programming books. Not one book talked about AVI files. I then read everything there was to read about the AVI format in the MSDN. Lots of useful information in the MSDN, but I needed more information.

After browsing the net for hours searching for AVI examples, I had just two sites bookmarked. I'm not going to say my search engine skills are amazing, but 99.9% of the time I have no problems finding what I'm looking for. I was absolutely shocked when I realized just how few AVI examples there were! Most the examples I found wouldn't compile... A handful of them were way to complex (for me at least), and the rest did the job, but they were coded in VB, Delphi, etc. (not VC++).

The first page I book marked was an article written by Jonathan Nix titled "AVI Files". You can visit it at http://www.gamedev.net/reference/programming/features/avifile/. Huge respect to Jonathan for writing an extremely brilliant document on the AVI format. Although I decided to do things differently, his example code snippets, and clear comments made the learning process alot easier! The second site is titled "The AVI Overview" by John F. McGowan, Ph.D.. I could go on and on about how amazing John's page is, but it's easier if you check it out yourself! The URL is http://www.jmcgowan.com/avi.html. His site pretty much covers everything there is to know about the AVI format! Thanks to John for making such a valuable page available to the public.

The last thing I wanted to mention is that NONE of the code has been borrowed, and none of the code has been copied. It was written during a 3 day coding spree, using information from the above mentioned sites and articles. With that said, I feel it is important to note that my code may not be the BEST way to play an AVI file. It may not even be the correct way to play an AVI file, but it does work, and it's easy to use! If you dislike the code, my coding style, or if you feel I'm hurting the programming community by releasing this tut, you have a few options: 1) search the net for alternate resources 2) write your own AVI player OR 3) write a better tutorial! Everyone visiting this site should know by now that I'm an average programmer with average skills (I've stated that on numerous pages throughout the site)! I code for FUN! The goal of this site is to make life easier for the non-elite coder to get started with OpenGL. The tutorials are merely examples on how 'I' managed to accomplish a specific effect... Nothing more, nothing less!

On to the code...

The first thing you will notice is that we include and link to the Video For Windows header / library. Big thanks to Microsoft (I can't believe I just said that!). This library makes opening and playing AVI files a SNAP! For now... All you need to know is that you MUST include the vfw.h header file and you must link to the vfw32.lib library file if you want the code to compile :)

```
#include <windows.h> // Header File For Windows
#include <gl\gl.h> // Header File For The OpenGL32 Library
#include <gl\glu.h> // Header File For The GLu32 Library
#include <vfw.h> // Header File For Video For Windows
#include "NeHeGL.h" // Header File For NeHeGL

#pragma comment( lib, "opengl32.lib" ) // Search For OpenGL32.lib While Linking
#pragma comment( lib, "glu32.lib" ) // Search For GLu32.lib While Linking
#pragma comment( lib, "vfw32.lib" ) // Search For VFW32.lib While Linking

#ifndef CDS_FULLSCREEN // CDS_FULLSCREEN Is Not Defined By Some
#define CDS_FULLSCREEN 4 // Compilers. By Defining It This Way,
#endif // We Can Avoid Errors
```

```
GL_Window* g_window;
Keys* g_keys;
```

Now we define our variables. angle is used to rotate our objects around based on the amount of time that has passed. We will use angle for all rotations just to keep things simple.

next is an integer variable that will be used to count how much time has passed (in milliseconds). It will be used to keep the framerate at a descent speed. More about this later!

frame is of course the current frame we want to display from the animation. We start off at 0 (first frame). I think it's safe to assume that if we managed to open the video, it HAS to have at least one frame of animation :)

effect is the current effect seen on the screen (object: Cube, Sphere, Cylinder, Nothing). env is a boolean value. If it's true, then environment mapping is enabled, if it's false, the object will NOT be environment mapped. If bg is true, you will see the video playing fullscreen behind the object. If it's false, you will only see the object (there will be no background).

sp, ep and bp are used to make sure the user isn't holding a key down.

```
// User Defined Variables
float angle; // Used For Rotation
int next; // Used For Animation
int frame=0; // Frame Counter
int effect; // Current Effect
bool sp; // Space Bar Pressed?
bool env=TRUE; // Environment Mapping (Default On)
bool ep; // 'E' Pressed?
bool bg=TRUE; // Background (Default On)
bool bp; // 'B' Pressed?
```

The psi structure will hold information about our AVI file later in the code. pavi is a pointer to a buffer that receives the new stream handle once the AVI file has been opened. pgf is a pointer to our GetFrame object. bmih will be used later in the code to convert the frame of animation to a format we want (holds the bitmap header info describing what we want). lastframe will hold the number of the last frame in the AVI animation. width and height will hold the dimensions of the AVI stream and finally.... pdata is a pointer to the image data returned after we get a frame of animation from the AVI! mpf will be used to calculate how many milliseconds each frame is displayed for. More on this later.

```
AVISTREAMINFO psi; // Pointer To A Structure Containing Stream Info
PAVISTREAM pavi; // Handle To An Open Stream
PGETFRAME pgf; // Pointer To A GetFrame Object
BITMAPINFOHEADER bmih; // Header Information For DrawDibDraw Decoding
long lastframe; // Last Frame Of The Stream
int width; // Video Width
int height; // Video Height
char *pdata; // Pointer To Texture Data
int mpf; // Will Hold Rough Milliseconds Per Frame
```

In this tutorial we will create 2 different quadratic shapes (a sphere and a cylinder) using the GLU library. quadratic is a pointer to our quadric object.

hdd is a handle to a DrawDib device context. hdc is handle to a device context.

hBitmap is a handle to a device dependant bitmap (used in the bitmap conversion process later).

data is a pointer that will eventually point to our converted bitmap image data. Will make sense later in the code. Keep reading :)

```
GLUquadricObj *quadratic; // Storage For Our Quadratic Objects

HDRAWDIB hdd; // Handle For Our Dib
HBITMAP hBitmap; // Handle To A Device Dependant Bitmap
HDC hdc = CreateCompatibleDC(0); // Creates A Compatible Device Context
unsigned char* data = 0; // Pointer To Our Resized Image
```

Now for some assembly language. For those of you that have never used assembly before, don't be intimidated. It might look cryptic, but it's actually pretty simple!

While writing this tutorial I discovered something very odd. The first video I actually got working with this code was playing fine but the colors were messed up. Everything that was supposed to be red was blue and everything that was supposed to be blue was red. I went absolutely NUTS! I was convinced that I made a mistake somewhere in the code. After looking at all the code, I was unable to find the bug! So I started reading through the MSDN again. Why would the red and blue bytes be swapped!?! It says right in the MSDN that 24 bit bitmaps are RGB!!! After some more reading I discovered the problem. In WINDOWS (figures), RGB data is actually store backwards (BGR). In OpenGL, RGB is exactly that... RGB!

After a few complaints from fans of Microsoft :) I decided to add a quick note! I am not trashing Microsoft because their RGB data is stored backwards. I just find it very frustrating that it's called RGB when it's actually BGR in the file!

Blue Adds: It's more to do with "little endian" and "big endian". Intel and Intel compatibles use little endian where the least significant byte (LSB) is stored first. OpenGL came from Silicon Graphics machines, which are probably big endian, and thus the OpenGL standard required the bitmap format to be in big endian format. I think this is how it works.

Wonderful! So here I am with a player, that looks like absolute crap! My first solution was to swap the bytes manually with a for next loop. It worked, but it was very slow. Completely fed up, I modified the texture generation code to use GL_BGR_EXT instead of GL_RGB. A huge speed increase, and the colors looked great! So my problem was solved... or so I thought! It turns out, some OpenGL drivers have problems with GL_BGR_EXT.... Back to the drawing board :(

After talking with my good friend Maxwell Sayles, he recommended that I swap the bytes using asm code. A minute later, he had icq'd me the code below! It may not be optimized, but it's fast and it does the job!

Each frame of animation is stored in a buffer. The image will always be 256 pixels wide, 256 pixels tall and 1 byte per color (3 bytes per pixel). The the code below will go through the buffer and swap the Red and Blue bytes. Red is stored at ebx+0 and blue is stored at ebx+2. We move through the buffer 3 bytes at a time (because one pixel is made up of 3 bytes). We loop through the data until all of the byte have been swapped.

A few of you were unhappy with the use of ASM code, so I figured I would explain why it's used in this tutorial. Originally I had planned to use GL_BGR_EXT as I stated, it works. But not on all cards! I then decided to use the swap method from the last tut (very tidy XOR swap code). The swap code works on all machines, but it's not extremely fast. In the last tut, yeah, it works GREAT. In this tutorial we are dealing with REAL-TIME video. You want the fastest swap you can get. Weighing the options, ASM in my opinion is the best choice! If you have a better way to do the job, please ... USE IT! I'm not telling you how you HAVE to do things. I'm showing you how I did it. I also explain in detail what the code does. That way if you want to replace the code with something better, you know exactly what this code is doing, making it easier to find an alternate solution if you want to write your own code!

```
void flipIt(void* buffer) // Flips The Red And Blue Bytes (256x256)
{
void* b = buffer; // Pointer To The Buffer
__asm // Assembler Code To Follow
{
mov ecx, 256*256 // Set Up A Counter (Dimensions Of Memory Block)
mov ebx, b // Points ebx To Our Data (b)
label: // Label Used For Looping
mov al,[ebx+0] // Loads Value At ebx Into al
mov ah,[ebx+2] // Loads Value At ebx+2 Into ah
mov [ebx+2],al // Stores Value In al At ebx+2
mov [ebx+0],ah // Stores Value In ah At ebx

add ebx,3 // Moves Through The Data By 3 Bytes
dec ecx // Decreases Our Loop Counter
jnz label // If Not Zero Jump Back To Label
}
}
```

The code below opens the AVI file in read mode. szFile is the name of the file we want to open. title[100] will be used to modify the title of the window (to show information about the AVI file).

The first thing we need to do is call AVIFileInit(). This initializes the AVI file library (gets things ready for us).

There are many ways to open an AVI file. I decided to use AVIStreamOpenFromFile(...). This opens a single stream from an AVI file (AVI files can contain multiple streams).

The parameters are as follows: pavi is a pointer to a buffer that receives the new stream handle. szFile is of course, the name of the file we wish to open (complete with path). The third parameter is the type of stream we wish to open. In this project, we are only interested in the VIDEO stream (streamtypeVIDEO). The fourth parameter is 0. This means we want the first occurance of streamtypeVIDEO (there can be multiple video streams in a single AVI file... we want the first stream). OF_READ means that we want to open the file for reading ONLY. The last parameter is a pointer to a class identifier of the handler you want to use. To be honest, I have no idea what it does. I let windows select it for me by passing NULL as the last parameter!

If there are any errors while opening the file, a message box pops up letting you know that the stream could not be opened. I don't pass a PASS or FAIL back to the calling section of code, so if this fails, the program will try to keep running. Adding some type of error checking shouldn't take alot of effort, I was too lazy :)

```
void OpenAVI(LPCSTR szFile) // Opens An AVI File (szFile)
{
TCHAR title[100]; // Will Hold The Modified Window Title

AVIFileInit(); // Opens The AVIFile Library

// Opens The AVI Stream
if (AVIStreamOpenFromFile(&pavi, szFile, streamtypeVIDEO, 0, OF_READ, NULL) !=0)
{
// An Error Occurred Opening The Stream
MessageBox (HWND_DESKTOP, "Failed To Open The AVI Stream", "Error", MB_OK | MB_ICONEXCLAMATION);
}
```

If we made it this far, it's safe to assume that the file was opened and a stream was located! Next we grab a bit of information from the AVI file with AVIStreamInfo(...).

Earlier we created a structure called psi that will hold information about our AVI stream. Will fill this structure with information about the AVI with the first line of code below. Everything from the width of the stream (in pixels) to the framerate of the animation is stored in psi. For those of you that want accurate playback speeds, make a note of what I just said. For more information look up AVIStreamInfo in the MSDN.

We can calculate the width of a frame by subtracting the left border from the right border. The result should be an accurate width in pixels. For the height, we subtract the top of the frame from the bottom of the frame. This gives us the height in pixels.

We then grab the last frame number from the AVI file using AVIStreamLength(...). This returns the number of frames of animation in the AVI file. The result is stored in lastframe.

Calculating the framerate is fairly easy. Frames per second = psi.dwRate / psi.dwScale. The value returned should match the frame rate displayed when you right click on the AVI and check its properties. So what does this have to do with mpf you ask? When I first wrote the animation code, I tried using the frames per second to select the correct frame of animation. I ran into a problem... All of the videos played to fast! So I had a look at the video properties. The face2.avi file is 3.36 seconds long. The frame rate is 29.974 frames per second. The video has 91 frames of animation. If you multiply 3.36 by 29.974 you get 100 frames of animation. Very Odd!

So, I decided to do things a little different. Instead of calculating the frames per second, I calculte how long each frame should be displayed. AVIStreamSampleToTime() converts a position in the animation to how many milliseconds it would take to get to that position. So we calculate how many milliseconds the entire video is by grabbing the time (in milliseconds) of the last frame (lastframe). We then divide the result by the total number of frames in the animation (lastframe). This gives us the amount of time each frame is displayed for in milliseconds. We store the result in mpf (milliseconds per frame). You could also calculate the milliseconds per frame by grabbing the amount of time for just 1 frame of animation with the following code: AVIStreamSampleToTime(pavi,1). Either way should work fine! Big thanks to Albert Chaulk for the idea!

The reason I say rough milliseconds per frame is because mpf is an integer so any floating values will be rounded off.

```
AVIStreamInfo(pavi, &psi, sizeof(psi)); // Reads Information About The Stream Into psi
width=psi.rcFrame.right-psi.rcFrame.left; // Width Is Right Side Of Frame Minus Left
height=psi.rcFrame.bottom-psi.rcFrame.top; // Height Is Bottom Of Frame Minus Top

lastframe=AVIStreamLength(pavi); // The Last Frame Of The Stream

mpf=AVIStreamSampleToTime(pavi,lastframe)/lastframe; // Calculate Rough Milliseconds Per Frame
```

Because OpenGL requires texture data to be a power of 2, and because most videos are 160x120, 320x240 or some other odd dimensions we need a fast way to resize the video on the fly to a format that we can use as a texture. To do this, we take advantage of specific Windows Dib functions.

The first thing we need to do is describe the type of image we want. To do this, we fill the bmih BitmapInfoHeader structure with our requested parameters. We start off by setting the size of the structure. We then set the bitplanes to 1. Three bytes of data works out to 24 bits (RGB). We want the image to be 256 pixels wide and 256 pixels tall and finally we want the data returned as UNCOMPRESSED RGB data (BI_RGB).

CreateDIBSection creates a dib that we can directly write to. If everything goes well, hBitmap will point to the dib's bit values. hdc is a handle to a device context (DC). The second parameter is a pointer to our BitmapInfo structure. The structure contains information about the dib file as mentioned above. The third parameter (DIB_RGB_COLORS) specifies that the data is RGB values. data is a pointer to a variable that receives a pointer to the location of the DIB's bit values (whew, that was a mouthful). By setting the 5th value to NULL, memory is allocated for our DIB. Finally, the last parameter can be ignored (set to NULL).

Quoted from the MSDN: The SelectObject function selects an object into the specified device context (DC).

We have now created a DIB that we can directly draw to. Yay :)

```
bmih.biSize = sizeof (BITMAPINFOHEADER); // Size Of The BitmapInfoHeader
bmih.biPlanes = 1; // Bitplanes
bmih.biBitCount = 24; // Bits Format We Want (24 Bit, 3 Bytes)
bmih.biWidth = 256; // Width We Want (256 Pixels)
bmih.biHeight = 256; // Height We Want (256 Pixels)
bmih.biCompression = BI_RGB; // Requested Mode = RGB

hBitmap = CreateDIBSection (hdc, (BITMAPINFO*)(&bmih), DIB_RGB_COLORS, (void**)(&data), NULL,
NULL);
SelectObject (hdc, hBitmap); // Select hBitmap Into Our Device Context (hdc)
```

A few more things to do before we're ready to read frames from the AVI. The next thing we have to do is prepare our program to decompress video frames from the AVI file. We do this with the AVIStreamGetFrameOpen(...) function.

You can pass a structure similar to the one above as the second parameter to have a specific video format returned. Unfortunately, the only thing you can alter is the width and height of the returned image. The MSDN also mentions that you can pass AVIGETFRAMEF_BESTDISPLAYFMT to select the best display format. Oddly enough, my compiler had no definition for it.

If everything goes well, a GETFRAME object is returned (which we need to read frames of data). If there are any problems, a message box will pop onto the screen telling you there was an error!

```
pgf=AVIStreamGetFrameOpen(pavi, NULL); // Create The PGETFRAME Using Our Request Mode
if (pgf==NULL)
{
// An Error Occurred Opening The Frame
MessageBox (HWND_DESKTOP, "Failed To Open The AVI Frame", "Error", MB_OK | MB_ICONEXCLAMATION);
}
```

The code below prints the videos width, height and frames to title. We display title at the top of the window with the command SetWindowText(...). Run the program in windowed mode to see what the code below does.

```
// Information For The Title Bar (Width / Height / Last Frame)
wsprintf (title, "NeHe's AVI Player: Width: %d, Height: %d, Frames: %d", width, height,
lastframe);
SetWindowText(g_window->hWnd, title); // Modify The Title Bar
}
```

Now for the fun stuff... we grab a frame from the AVI and then convert it to a usable image size / color depth. lpbi will hold the BitmapInfoHeader information for the frame of animation. We accomplish a few things at once in the second line of code below. First we grab a frame of animation ... The frame we want is specified by frame. This will pull in the frame of animation and will fill lpbi with the header information for that frame.

Now for the fun stuff... we need to point to the image data. To do this we need to skip over the header information (lpbi->biSize). One thing I didn't realize until I started writing this tut was that we also have to skip over any color information. To do this we also add colors used multiplied by the size of RGBQUAD (biClrUsed*sizeof(RGBQUAD)). After doing ALL of that :) we are left with a pointer to the image data (pdata).

Now we need to convert the frame of animation to a usuable texture size as well, we need to convert the data to RGB data. To do this, we use DrawDibDraw(...).

A quick explanation. We can draw directly to our custom DIB. That's what DrawDibDraw(...) does. The first parameter is a handle to our DrawDib DC. The second parameter is a handle to the DC. Next we have the upper left corner (0,0) and the lower right corner (256,256) of the destination rectangle.

lpbi is a pointer to the bitmapinfoheader information for the frame we just read. pdata is a pointer to the image data for the frame we just read.

Then we have the upper left corner (0,0) of the source image (frame we just read) and the lower right corner of the frame we just read (width of the frame, height of the frame). The last parameter should be left at 0.

This will convert an image of any size / color depth to a 256*256*24bit image.

```
void GrabAVIFrame(int frame) // Grabs A Frame From The Stream
{
LPBITMAPINFOHEADER lpbi; // Holds The Bitmap Header Information
lpbi = (LPBITMAPINFOHEADER)AVIStreamGetFrame(pgf, frame); // Grab Data From The AVI Stream
pdata=(char *)lpbi+lpbi->biSize+lpbi->biClrUsed * sizeof(RGBQUAD); // Pointer To Data Returned
By AVIStreamGetFrame
// (Skip The Header Info To Get To The Data)
// Convert Data To Requested Bitmap Format
DrawDibDraw (hdd, hdc, 0, 0, 256, 256, lpbi, pdata, 0, 0, width, height, 0);
```

We have our frame of animation but the red and blue bytes are swapped. To solve this problem, we jump to our speedy flipIt(...) code. Remember, data is a pointer to a variable that receives a pointer to the location of the DIB's bit values. What that means is that after we call DrawDibDraw, data will point to the resized (256*256) / modified (24 bit) bitmap data.

Originally I was updating the texture by recreating it for each frame of animation. I received a few emails suggesting that I use glTexSubImage2D(). After flipping through the OpenGL Red Book, I stumbled across the following quote: "Creating a texture may be more computationally expensive than modifying an existing one. In OpenGL Release 1.1, there are new routines to replace all or part of a texture image with new information. This can be helpful for certain applications, such as using real-time, captured video images as texture images. For that application, it makes sense to create a single texture and use glTexSubImage2D() to repeatedly replace the texture data with new video images".

I personally didn't notice a huge speed increase, but on slower cards you might! The parameters for glTexSubImage2D() are as follows: Our target, which is a 2D texture (GL_TEXTURE_2D). The detail level (0), used for mipmapping. The x (0) and y (0) offset which tells OpenGL where to start copying to (0,0 is the lower left corner of the texture). The we have the width of the image we wish to copy which is 256 pixels wide and 256 pixels tall. GL_RGB is the format of our data. We are copying unsigned bytes. Finally... The pointer to our data which is represented by data. Very simple!

Kevin Rogers Adds: I just wanted to point out another important reason to use glTexSubImage2D. Not only is it faster on many OpenGL implementations, but the target area does not need to be a power of 2. This is especially handy for video playback since the typical dimensions for a frame are rarely powers of 2 (often something like 320 x 200). This gives you the flexibility to play the video stream at its original aspect, rather than distorting / clipping each frame to fit your texture dimensions.

It's important to note that you can NOT update a texture if you have not created the texture in the first place! We create the texture in the Initialize() code!

I also wanted to mention... If you planned to use more than one texture in your project, make sure you bind the texture you want to update. If you don't bind the texture you may end up updating textures you didn't want updated!

```
flipIt(data); // Swap The Red And Blue Bytes (GL Compatability)

// Update The Texture
glTexSubImage2D (GL_TEXTURE_2D, 0, 0, 0, 256, 256, GL_RGB, GL_UNSIGNED_BYTE, data);
```

```
}
```

The following section of code is called when the program exits. We close our DrawDib DC, and free allocated resources. We then release the AVI GetFrame resources. Finally we release the stream and then the file.

```
void CloseAVI(void) // Properly Closes The Avi File
{
DeleteObject(hBitmap); // Delete The Device Dependant Bitmap Object
DrawDibClose(hdd); // Closes The DrawDib Device Context
AVIStreamGetFrameClose(pgf); // Deallocates The GetFrame Resources
AVIStreamRelease(pavi); // Release The Stream
AVIFileExit(); // Release The File
}
```

Initialization is pretty straight forward. We set the starting angle to 0. We then open the DrawDib library (which grabs a DC). If everything goes well, hdd becomes a handle to the newly created device context.

Our clear screen color is black, depth testing is enabled, etc.

We then create a new quadric. quadratic is the pointer to our new object. We set up smooth normals, and enable texture coordinate generation for our quadric.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Any GL Init Code & User Initialiazation Goes
Here
{
g_window = window;
g_keys = keys;

// Start Of User Initialization
angle = 0.0f; // Set Starting Angle To Zero
hdd = DrawDibOpen(); // Grab A Device Context For Our Dib
glClearColor (0.0f, 0.0f, 0.0f, 0.5f); // Black Background
glClearDepth (1.0f); // Depth Buffer Setup
glDepthFunc (GL_LEQUAL); // The Type Of Depth Testing (Less Or Equal)
glEnable(GL_DEPTH_TEST); // Enable Depth Testing
glShadeModel (GL_SMOOTH); // Select Smooth Shading
glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Set Perspective Calculations To Most
Accurate

quadratic=gluNewQuadric(); // Create A Pointer To The Quadric Object
gluQuadricNormals(quadratic, GLU_SMOOTH); // Create Smooth Normals
gluQuadricTexture(quadratic, GL_TRUE); // Create Texture Coords
```

In the next bit of code, we enable 2D texture mapping, we set the texture filters to GL_NEAREST (fast, but rough looking) and we set up sphere mapping (to create the environment mapping effect). Play around with the filters. If you have the power, try out GL_LINEAR for a smoother looking animation.

After setting up our texture and sphere mapping, we open the .AVI file. I tried to keep things simple... can you tell :) The file we are going to open is called face2.avi... it's located in the data directory.

The last thing we have to do is create our initial texture. We need to do this in order to use glTexSubImage2D() to update our texture in GrabAVIFrame().

```
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);// Set Texture Max Filter
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);// Set Texture Min Filter

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set The Texture Generation Mode For S To
Sphere Mapping
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Set The Texture Generation Mode For T To
Sphere Mapping

OpenAVI("data/face2.avi"); // Open The AVI File

// Create The Texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE, data);

return TRUE; // Return TRUE (Initialization Successful)
}
```

When shutting down, we call CloseAVI(). This properly closes the AVI file, and releases any used resources.

```
void Deinitialize (void) // Any User DeInitialization Goes Here
{
CloseAVI(); // Close The AVI File
}
```

This is where we check for key presses and update our rotation (angle) based on time passed. By now I shouldn't have to explain the code in detail. We check to see if the space bar is pressed. If it is, we increase the effect. We have three effect (cube, sphere, cylinder) and when the 4th effect is selected (effect=3) nothing is drawn... showing just the background scene! If we are on the 4th effect and space is pressed, we reset back to the first effect (effect=0). Yeah, I know I should have called it OBJECT :)

We then check to see if the 'B' key is pressed if it is, we toggle the background (bg) from ON to OFF or from OFF to ON.

Environment mapping is done the same way. We check to see if 'E' is pressed. If it is, we toggle env from TRUE to FALSE or from FALSE to TRUE. Turning environment mapping off or on!

The angle is increased by a tiny fraction each time Update() is called. I divide the time passed by 60.0f to slow things down a little.

```
void Update (DWORD milliseconds) // Perform Motion Updates Here
{
if (g_keys->keyDown [VK_ESCAPE] == TRUE) // Is ESC Being Pressed?
{
TerminateApplication (g_window); // Terminate The Program
}

if (g_keys->keyDown [VK_F1] == TRUE) // Is F1 Being Pressed?
{
ToggleFullscreen (g_window); // Toggle Fullscreen Mode
}

if ((g_keys->keyDown [' ']) && !sp) // Is Space Being Pressed And Not Held?
{
sp=TRUE; // Set sp To True
effect++; // Change Effects (Increase effect)
if (effect>3) // Over Our Limit?
effect=0; // Reset Back To 0
}

if (!g_keys->keyDown[' ']) // Is Space Released?
sp=FALSE; // Set sp To False

if ((g_keys->keyDown ['B']) && !bp) // Is 'B' Being Pressed And Not Held?
{
bp=TRUE; // Set bp To True
bg=!bg; // Toggle Background Off/On
}

if (!g_keys->keyDown['B']) // Is 'B' Released?
bp=FALSE; // Set bp To False

if ((g_keys->keyDown ['E']) && !ep) // Is 'E' Being Pressed And Not Held?
{
ep=TRUE; // Set ep To True
env=!env; // Toggle Environment Mapping Off/On
}

if (!g_keys->keyDown['E']) // Is 'E' Released?
ep=FALSE; // Set ep To False

angle += (float)(milliseconds) / 60.0f; // Update angle Based On The Timer
```

In the original tutorial, all AVI files were played at the same speed. Since then, the tutorial has been rewritten to play the video at the correct speed. next is increased by the number of milliseconds that have passed since this section of code was last called. If you remember earlier in the tutorial, we calculated how long each frame should be displayed in milliseconds (mpf). To calculate the current frame, we take the amount of time that has passed (next) and divide it by the time each frame is displayed for (mpf).

After that, we check to make sure that the current frame of animation hasn't passed the last frame of the video. If it has, frame is reset to zero, the animation timer (next) is reset to 0, and the animation starts over.

The code below will drop frames if your computer is running to slow, or another application is hogging the CPU. If you want every frame to be displayed no matter how slow the users computer is, you could check to see if next is greater than mpf if it is, you would reset next to 0 and increase frame by one. Either way will work, although the code below is better for faster machines.

If you feel energetic, try adding rewind, fast forward, pause or reverse play!

```
next+= milliseconds; // Increase next Based On Timer (Milliseconds)
frame=next/mpf; // Calculate The Current Frame

if (frame>=lastframe) // Have We Gone Past The Last Frame?
{
frame=0; // Reset The Frame Back To Zero (Start Of Video)
next=0; // Reset The Animation Timer (next)
}
}
```

Now for the drawing code :) We clear the screen and depth buffer. We then grab a frame of animation. Again, I tried to keep it simple! You pass the requested frame (frame) to GrabAVIFrame(). Pretty simple! Of course, if you wanted multiple AVI's, you would have to pass a texture ID. (More for you to do).

```
void Draw (void) // Draw Our Scene
{
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer

GrabAVIFrame(frame); // Grab A Frame From The AVI
```

The code below checks to see if we want to draw a background image. If bg is TRUE, we reset the modelview matrix and draw a single texture mapped quad (mapped with a frame from the AVI video) large enough to fill the entire screen. The quad is drawn 20 units into the screen so it appears behind the object (futher in the distance).

```
if (bg) // Is Background Visible?
{
glLoadIdentity(); // Reset The Modelview Matrix
glBegin(GL_QUADS); // Begin Drawing The Background (One Quad)
// Front Face
glTexCoord2f(1.0f, 1.0f); glVertex3f( 11.0f, 8.3f, -20.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-11.0f, 8.3f, -20.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-11.0f, -8.3f, -20.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 11.0f, -8.3f, -20.0f);
glEnd(); // Done Drawing The Background
}
```

After drawing the background (or not), we reset the modelview matrix (starting us back at the center of the screen). We then translate 10 units into the screen.

After that, we check to see if env is TRUE. If it is, we enable sphere mapping to create the environment mapping effect.

```
glLoadIdentity (); // Reset The Modelview Matrix
glTranslatef (0.0f, 0.0f, -10.0f); // Translate 10 Units Into The Screen

if (env) // Is Environment Mapping On?
{
glEnable(GL_TEXTURE_GEN_S); // Enable Texture Coord Generation For S (NEW)
glEnable(GL_TEXTURE_GEN_T); // Enable Texture Coord Generation For T (NEW)
}
```

I added the code below at the last minute. It rotates on the x-axis and y-axis (based on the value of angle) and then translates 2 units on the z-axis. This move us away from the center of the screen. If you remove the three lines of code below, the object will spin in the center of the screen. With the three lines of code, the objects move around a bit as they spin :)

If you don't understand rotations and translations... you shouldn't be reading this tutorial :)

```
glRotatef(angle*2.3f,1.0f,0.0f,0.0f); // Throw In Some Rotations To Move Things Around A Bit
glRotatef(angle*1.8f,0.0f,1.0f,0.0f); // Throw In Some Rotations To Move Things Around A Bit
glTranslatef(0.0f,0.0f,2.0f); // After Rotating Translate To New Position
```

The code below checks to see which effect (object) we want to draw. If the value of effect is 0, we do a few rotations and then draw a cube. The rotations keep the cube spinning on the x-axis, y-axis and z-axis. By now, you should have the code to create a cube burned into your head :)

```
switch (effect) // Which Effect?
{
case 0: // Effect 0 - Cube
glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f); // Rotate On The X-Axis By angle
glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f); // Rotate On The Y-Axis By angle
glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f); // Rotate On The Z-Axis By angle
glBegin(GL_QUADS); // Begin Drawing A Cube
// Front Face
glNormal3f( 0.0f, 0.0f, 0.5f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
// Back Face
glNormal3f( 0.0f, 0.0f,-0.5f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Top Face
glNormal3f( 0.0f, 0.5f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
// Bottom Face
glNormal3f( 0.0f,-0.5f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
// Right Face
glNormal3f( 0.5f, 0.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Left Face
glNormal3f(-0.5f, 0.0f, 0.0f);
```

```
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd(); // Done Drawing Our Cube
break; // Done Effect 0
```

This is where we draw the sphere. We start off with a few quick rotations on the x-axis, y-axis and z-axis. We then draw the sphere. The sphere will have a radius of 1.3f, with 20 slices and 20 stacks. I decided to use 20 because I didn't want the sphere to be perfectly smooth. Using fewer slices and stacks gives the sphere a rougher look (less smooth), making it semi obvious that the sphere is actually rotating when sphere mapping is enabled. Try playing around with the values! It's important to note that more slices or stacks requires more processing power!

```
case 1: // Effect 1 - Sphere
glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f); // Rotate On The X-Axis By angle
glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f); // Rotate On The Y-Axis By angle
glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f); // Rotate On The Z-Axis By angle
gluSphere(quadratic,1.3f,20,20); // Draw A Sphere
break; // Done Drawing Sphere
```

This is where we draw the cylinder. We start off with some simple rotations on the x-axis, y-axis and z-axis. Our cylinder has a base and top radius of 1.0f units. It's 3.0f units high, and is composed of 32 slices and 32 stacks. If you decrease the slices or stacks, the cylinder will be made up of less polygons and will appear less rounded.

Before we draw the cylinder, we translate -1.5f units on the z-axis. By doing this, our cylinder will rotate around it's center point. The general rule to centering a cylinder is to divide it's height by 2 and translate by the result in a negative direction on the z-axis. If you have no idea what I'm talking about, take out the tranlatef(...) line below. The cylinder will rotate around it's base, instead of a center point.

```
case 2: // Effect 2 - Cylinder
glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f); // Rotate On The X-Axis By angle
glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f); // Rotate On The Y-Axis By angle
glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f); // Rotate On The Z-Axis By angle
glTranslatef(0.0f,0.0f,-1.5f); // Center The Cylinder
gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // Draw A Cylinder
break; // Done Drawing Cylinder
}
```

Next we check to see if env is TRUE. If it is, we disable sphere mapping. We call glFlush() to flush out the rendering pipeline (makes sure everything gets rendered before we draw the next frame).

```
if (env) // Environment Mapping Enabled?
{
glDisable(GL_TEXTURE_GEN_S); // Disable Texture Coord Generation For S (NEW)
glDisable(GL_TEXTURE_GEN_T); // Disable Texture Coord Generation For T (NEW)
}

glFlush (); // Flush The GL Rendering Pipeline
}
```

I hope you enjoyed this tutorial. It's 2:00am at the moment... I've been working on this tut for the last 6 hours. Sounds crazy, but writing things so that they actually make sense is not an easy task. I have read the tut 3 times now and I'm still trying to make things easier to understand. Believe it or not, it's important to me that you understand how things work and why they work. That's why I babble endlessly, why I over-comment, etc.

Anyways... I would love to hear some feedback about this tut. If you find mistakes or you would like to help make the tut better, please contact me. As I said, this is my first attempt at AVI. Normally I wouldn't write a tut on a subject I just learned, but my excitement got the best of me, plus the fact that there's very little information on the subject bothered me. What I'm hoping is that I'll open the door to a flood of higher quality AVI demos and example code! Might happen... might not. Either way, the code is here for you to use however you want!

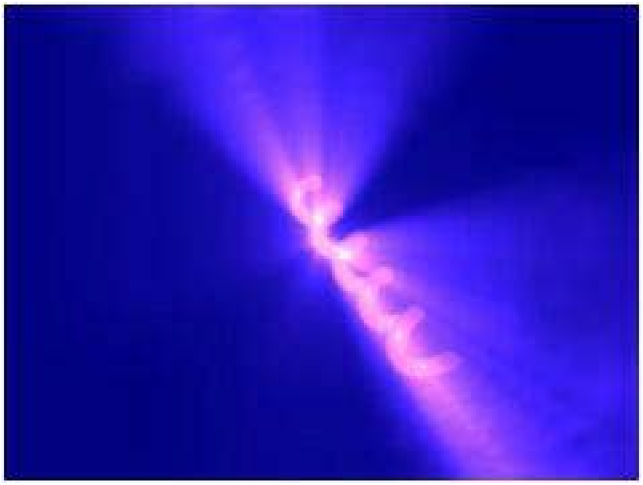
Huge thanks to Fredster for the face AVI file. Face was one of about 6 AVI animations he sent to me for use in my tutorial. No questions asked, no conditions. I emailed him and he went out of his way to help me out... Huge respect!

An even bigger thanks to Jonathan de Blok. If it wasn't for him, this tutorial would not exist. He got me interested in the AVI format by sending me bits of code from his own personal AVI player. He also went out of his way to answer any questions that I had in regards to his code. It's important to note that nothing was borrowed or taken from his code, it was used only to understand how an AVI player works. My player opens, decodes and plays AVI files using very different code!

Thanks to everyone for the great support! This site would be nothing without it's visitors!!!

Jeff Molofee (**NeHe**)

Lesson 36
Radial Blur & Rendering To A Texture



Hi! I'm Dario Corno, also known as rIo of SpinningKids. First of all, I want to explain why I decided to write this little tutorial. I have been a scener since 1989. I want all of you to download some demos so you understand what a demo is and what demo effects are.

Demos are done to show off hardcore and sometimes brutal coding as well as artistic skill. You can usually find some really killer effects in todays demos! This won't be a killer effect tutorial, but the end result is very cool! You can find a huge collection of demos at http://www.pouet.net and http://ftp.scene.org.

Now that the introduction is out of the way, we can go on with the tutorial...

I will explain how to do an eye candy effect (used in demos) that looks like radial blur. Sometimes it's referred to as volumetric lights, don't believe it, it's just a fake radial blur! ;D

Radial blur was usually done (when there were only software renderers) by blurring every pixel of the original image in a direction opposite the center of the blur.

With todays hardware it is quite difficult to do blurring by hand using the color buffer (at least in a way that is supported by all the gfx cards), so we need to do a little trick to achieve the same effect.

As a bonus while learning the radial blur effect, you will also learn how to render to a texture the easy way!

I decided to use a spring as the shape in this tutorial because it's a cool shape, and I'm tired of cubes :)

It's important to note that this tutorial is more a guideline on how to create the effect. I don't go into great detail explaining the code. You should know most of it off by heart :)

Below are the variable definitions and includes used:

```
#include <math.h> // We'll Need Some Math

float angle; // Used To Rotate The Helix
float vertexes[4][3]; // Holds Float Info For 4 Sets Of Vertices
float normal[3]; // An Array To Store The Normal Data
GLuint BlurTexture; // An Unsigned Int To Store The Texture Number
```

The function EmptyTexture() creates an empty texture and returns the number of that texture. We just allocate some free space (exactly 128 * 128 * 4 unsiged integers).

128 * 128 is the size of the texture (128 pixels wide and tall), the 4 means that for every pixel we want 4 byte to store the RED, GREEN, BLUE and ALPHA components.

```
GLuint EmptyTexture() // Create An Empty Texture
```

```
{
GLuint txtnumber; // Texture ID
unsigned int* data; // Stored Data

// Create Storage Space For Texture Data (128x128x4)
data = (unsigned int*)new GLuint[((128 * 128)* 4 * sizeof(unsigned int))];
```

After allocating space we zero it using the ZeroMemory function, passing the pointer (data) and the size of memory to be "zeroed".

A semi important thing to note is that we set the magnification and minification methods to GL_LINEAR. That's because we will be stretching our texture and GL_NEAREST looks quite bad if stretched.

```
ZeroMemory(data,((128 * 128)* 4 * sizeof(unsigned int))); // Clear Storage Memory

glGenTextures(1, &txtnumber); // Create 1 Texture
glBindTexture(GL_TEXTURE_2D, txtnumber); // Bind The Texture
glTexImage2D(GL_TEXTURE_2D, 0, 4, 128, 128, 0,
GL_RGBA, GL_UNSIGNED_BYTE, data); // Build Texture Using Information In data
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

delete [] data; // Release data

return txtnumber; // Return The Texture ID
}
```

This function simply normalize the length of the normal vectors. Vectors are expressed as arrays of 3 elements of type float, where the first element rapresent X, the second Y and the third Z. A normalized vector (Nv) is expressed by Vn = (Vox / |Vo| , Voy / |Vo|, Voz / |Vo|), where Vo is the original vector, |vo| is the modulus (or length) of that vector, and x,y,z are the component of that vector. Doing it "digitally" will be: Calculating the length of the original vector by doing: sqrt(x^2 + y^2 + z^2) ,where x,y,z are the 3 components of the vector. And then dividing each normal vector component by the length of the vector.

```
void ReduceToUnit(float vector[3]) // Reduces A Normal Vector (3 Coordinates)
{ // To A Unit Normal Vector With A Length Of One.
float length; // Holds Unit Length
// Calculates The Length Of The Vector
length = (float)sqrt((vector[0]*vector[0]) + (vector[1]*vector[1]) + (vector[2]*vector[2]));

if(length == 0.0f) // Prevents Divide By 0 Error By Providing
length = 1.0f; // An Acceptable Value For Vectors To Close To 0.

vector[0] /= length; // Dividing Each Element By
vector[1] /= length; // The Length Results In A
vector[2] /= length; // Unit Normal Vector.
}
```

The following routine calculates the normal given 3 vertices (always in the 3 float array). We have two parameters : v[3][3] and out[3] of course the first parameter is a matrix of floats with m=3 and n=3 where every line is a vertex of the triangle. out is the place where we'll put the resulting normal vector.

A bit of (easy) math. We are going to use the famous cross product, by definition the cross product is an operation between two vectors that returns another vector orthogonal to the two original vectors. The normal is the vector orthogonal to a surface, with the versus opposite to that surface (and usually a normalized length). Imagine now if the two vectors above are the sides of a triangle, then the orthogonal vector (calculated with the cross product) of two sides of a triangle is exactly the normal of that triangle.

Harder to explain than to do.

We will start finding the vector going from vertex 0 to vertex 1, and the vector from vertex 1 to vertex 2, this is basically done by (brutally) subtracting each component of each vertex from the next. Now we got the vectors for our triangle sides. By doing the cross product (vXw) we get the normal vector for that triangle.

Let's see the code.

v[0][] is the first vertex, v[1][] is the second vertex, v[2][] is the third vertex. Every vertex has: v[][0] the x coordinate of that vertex, v[][1] the y coord of that vertex, v[][2] the z coord of that vertex.

By simply subtracting every coord of one vertex from the next we get the VECTOR from this vertex to the next. v1[0] = v[0][0] - v[1][0], this calculates the X component of the VECTOR going from VERTEX 0 to vertex 1. v1[1] = v[0][1] - v[1][1], this will calculate the Y component v1[2] = v[0][2] - v[1][2], this will calculate the Z component and so on...

Now we have the two VECTORS, so let's calculate the cross product of them to get the normal of the triangle.

The formula for the cross product is:

out[x] = v1[y] * v2[z] - v1[z] * v2[y]

out[y] = v1[z] * v2[x] - v1[x] * v2[z]

out[z] = v1[x] * v2[y] - v1[y] * v2[x] We finally have the normal of the triangle in out[].

```
void calcNormal(float v[3][3], float out[3]) // Calculates Normal For A Quad Using 3 Points
{
float v1[3],v2[3]; // Vector 1 (x,y,z) & Vector 2 (x,y,z)
static const int x = 0; // Define X Coord
static const int y = 1; // Define Y Coord
static const int z = 2; // Define Z Coord

// Finds The Vector Between 2 Points By Subtracting
// The x,y,z Coordinates From One Point To Another.

// Calculate The Vector From Point 1 To Point 0
v1[x] = v[0][x] - v[1][x]; // Vector 1.x=Vertex[0].x-Vertex[1].x
v1[y] = v[0][y] - v[1][y]; // Vector 1.y=Vertex[0].y-Vertex[1].y
v1[z] = v[0][z] - v[1][z]; // Vector 1.z=Vertex[0].y-Vertex[1].z
// Calculate The Vector From Point 2 To Point 1
v2[x] = v[1][x] - v[2][x]; // Vector 2.x=Vertex[0].x-Vertex[1].x
v2[y] = v[1][y] - v[2][y]; // Vector 2.y=Vertex[0].y-Vertex[1].y
v2[z] = v[1][z] - v[2][z]; // Vector 2.z=Vertex[0].z-Vertex[1].z
// Compute The Cross Product To Give Us A Surface Normal
out[x] = v1[y]*v2[z] - v1[z]*v2[y]; // Cross Product For Y - Z
out[y] = v1[z]*v2[x] - v1[x]*v2[z]; // Cross Product For X - Z
out[z] = v1[x]*v2[y] - v1[y]*v2[x]; // Cross Product For X - Y

ReduceToUnit(out); // Normalize The Vectors
}
```

The following routine just sets up a point of view using gluLookAt. We set a point of view placed at 0, 5, 50 that is looking to 0, 0, 0 and that has the UP vector looking UP (0, 1, 0)! :D

```
void ProcessHelix() // Draws A Helix
{
GLfloat x; // Helix x Coordinate
GLfloat y; // Helix y Coordinate
GLfloat z; // Helix z Coordinate
GLfloat phi; // Angle
GLfloat theta; // Angle
GLfloat v,u; // Angles
GLfloat r; // Radius Of Twist
int twists = 5; // 5 Twists

GLfloat glfMaterialColor[]={0.4f,0.2f,0.8f,1.0f}; // Set The Material Color
GLfloat specular[]={1.0f,1.0f,1.0f,1.0f}; // Sets Up Specular Lighting

glLoadIdentity(); // Reset The Modelview Matrix
gluLookAt(0, 5, 50, 0, 0, 0, 0, 1, 0); // Eye Position (0,5,50) Center Of Scene (0,0,0)
// Up On Y Axis.
glPushMatrix(); // Push The Modelview Matrix

glTranslatef(0,0,-50); // Translate 50 Units Into The Screen
glRotatef(angle/2.0f,1,0,0); // Rotate By angle/2 On The X-Axis
glRotatef(angle/3.0f,0,1,0); // Rotate By angle/3 On The Y-Axis

glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,glfMaterialColor);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,specular);
```

We then calculate the helix formula and render the spring. It's quite simple, I won't explain it, beacuse it isn't the main goal of this tutorial. The helix code was borrowed (and optimized a bit) from Listen Software friends. This is written the simple way, and is not the fastest method. Using vertex arrays would make it faster!

```
r=1.5f; // Radius

glBegin(GL_QUADS); // Begin Drawing Quads
for(phi=0; phi <= 360; phi+=20.0) // 360 Degrees In Steps Of 20
{
for(theta=0; theta<=360*twists; theta+=20.0) // 360 Degrees * Number Of Twists In Steps Of 20
{
v=(phi/180.0f*3.142f); // Calculate Angle Of First Point ( 0 )
u=(theta/180.0f*3.142f); // Calculate Angle Of First Point ( 0 )

x=float(cos(u)*(2.0f+cos(v) ))*r; // Calculate x Position (1st Point)
y=float(sin(u)*(2.0f+cos(v) ))*r; // Calculate y Position (1st Point)
z=float((( u-(2.0f*3.142f)) + sin(v) ) * r); // Calculate z Position (1st Point)

vertexes[0][0]=x; // Set x Value Of First Vertex
vertexes[0][1]=y; // Set y Value Of First Vertex
vertexes[0][2]=z; // Set z Value Of First Vertex

v=(phi/180.0f*3.142f); // Calculate Angle Of Second Point ( 0 )
u=((theta+20)/180.0f*3.142f); // Calculate Angle Of Second Point ( 20 )

x=float(cos(u)*(2.0f+cos(v) ))*r; // Calculate x Position (2nd Point)
y=float(sin(u)*(2.0f+cos(v) ))*r; // Calculate y Position (2nd Point)
z=float((( u-(2.0f*3.142f)) + sin(v) ) * r); // Calculate z Position (2nd Point)
vertexes[1][0]=x; // Set x Value Of Second Vertex
vertexes[1][1]=y; // Set y Value Of Second Vertex
vertexes[1][2]=z; // Set z Value Of Second Vertex

v=((phi+20)/180.0f*3.142f); // Calculate Angle Of Third Point ( 20 )
u=((theta+20)/180.0f*3.142f); // Calculate Angle Of Third Point ( 20 )

x=float(cos(u)*(2.0f+cos(v) ))*r; // Calculate x Position (3rd Point)
y=float(sin(u)*(2.0f+cos(v) ))*r; // Calculate y Position (3rd Point)
z=float((( u-(2.0f*3.142f)) + sin(v) ) * r); // Calculate z Position (3rd Point)

vertexes[2][0]=x; // Set x Value Of Third Vertex
vertexes[2][1]=y; // Set y Value Of Third Vertex
vertexes[2][2]=z; // Set z Value Of Third Vertex

v=((phi+20)/180.0f*3.142f); // Calculate Angle Of Fourth Point ( 20 )
u=((theta)/180.0f*3.142f); // Calculate Angle Of Fourth Point ( 0 )

x=float(cos(u)*(2.0f+cos(v) ))*r; // Calculate x Position (4th Point)
y=float(sin(u)*(2.0f+cos(v) ))*r; // Calculate y Position (4th Point)
z=float((( u-(2.0f*3.142f)) + sin(v) ) * r); // Calculate z Position (4th Point)

vertexes[3][0]=x; // Set x Value Of Fourth Vertex
vertexes[3][1]=y; // Set y Value Of Fourth Vertex
vertexes[3][2]=z; // Set z Value Of Fourth Vertex

calcNormal(vertexes,normal); // Calculate The Quad Normal

glNormal3f(normal[0],normal[1],normal[2]); // Set The Normal

// Render The Quad
glVertex3f(vertexes[0][0],vertexes[0][1],vertexes[0][2]);
glVertex3f(vertexes[1][0],vertexes[1][1],vertexes[1][2]);
glVertex3f(vertexes[2][0],vertexes[2][1],vertexes[2][2]);
glVertex3f(vertexes[3][0],vertexes[3][1],vertexes[3][2]);
}
}
glEnd(); // Done Rendering Quads

glPopMatrix(); // Pop The Matrix
}
```

This two routines (ViewOrtho and ViewPerspective) were coded to make it easy to draw in an orthogonal way and get back to perspective rendering with ease.

ViewOrtho simply sets the projection matrix, then pushes a copy of the actual projection matrix onto the OpenGL stack. The identity matrix is then loaded and an orthographic view with the current screen resolution is set up.

This way it is possible to draw using 2D coordinates with 0,0 in the upper left corner of the screen and with 640,480 in the lower right corner of the screen.

Finally, the modelview matrix is activated for rendering stuff.

ViewPerspective sets up projection matrix mode and pops back the non-orthogonal matrix that ViewOrtho pushed onto the stack. The modelview matrix is then selected so we can rendering stuff.

I suggest you keep these two procedures, it's nice being able to render in 2D without having to worry about the projection matrix!

```
void ViewOrtho() // Set Up An Ortho View
{
glMatrixMode(GL_PROJECTION); // Select Projection
glPushMatrix(); // Push The Matrix
glLoadIdentity(); // Reset The Matrix
glOrtho( 0, 640 , 480 , 0, -1, 1 ); // Select Ortho Mode (640x480)
glMatrixMode(GL_MODELVIEW); // Select Modelview Matrix
glPushMatrix(); // Push The Matrix
glLoadIdentity(); // Reset The Matrix
}

void ViewPerspective() // Set Up A Perspective View
{
glMatrixMode( GL_PROJECTION ); // Select Projection
glPopMatrix(); // Pop The Matrix
glMatrixMode( GL_MODELVIEW ); // Select Modelview
glPopMatrix(); // Pop The Matrix
}
```

Now it's time to explain how the fake radial blur effect is done:

We need to draw the scene so it appears blurred in all directions starting from the center. The trick is doing this without a major performance hit. We can't read and write pixels, and if we want compatibility with non kick-butt video cards, we can't use extensions or driver specific commands.

Time to give up... ?

No, the solution is quite easy, OpenGL gives us the ability to "blur" textures. Ok... Not really blurring, but if we scale a texture using linear filtering, the result (with a bit of imagination) looks like gaussian blur.

So what would happen if we put a lot of stretched textures right on top of the 3D scene and scaled them?

The answer is simple... A radial blur effect!

There are two problems: How do we create the texture realtime and how do we place the texture exactly in front of the 3D object?

The solutions are easier than you may think!

Problem ONE: Rendering To A Texture

The problem is easy to solve on pixel formats that have a back buffer. Rendering to a texture without a back buffer can be a real pain on the eyes!

Rendering to texture is achieved with just one function! We need to draw our object and then copy the result (BEFORE SWAPPING THE BACK BUFFER WITH THE FRONT BUFFER) to a texture using the glCopytexSubImage function.

Problem TWO: Fitting The Texture Exactly In Front Of The 3D Object

We know that, if we change the viewport without setting the right perspective, we get a stretched rendering of our object. For example if we set a viewport really wide we get a vertically stretched rendering.

The solution is first to set a viewport that is square like our texture (128x128). After rendering our object to the texture, we render the texture to the screen using the current screen resolution. This way OpenGL reduces the object to fit into the texture, and when we stretch the texture to the full size of the screen, OpenGL resizes the texture to fit perfectly over top of our 3d object. Hopefully I haven't lost anyone. Another quick example... If you took a 640x480 screenshot, and then resized the screenshot to a 256x256 bitmap, you could load that bitmap as a texture and stretch it to fit on a 640x480 screen. The quality would not be as good, but the texture should line up pretty close to the original 640x480 image.

On to the fun stuff! This function is really easy and is one of my preferred "design tricks". It sets a viewport with a size that matches our BlurTexture dimensions (128x128). It then calls the routine that renders the spring. The spring will be stretched to fit the 128*128 texture because of the viewport (128x128 viewport).

After the spring is rendered to fit the 128x128 viewport, we bind to the BlurTexture and copy the colour buffer from the viewport to the BlurTexture using glCopyTexSubImage2D.

The parameters are as follows:

GL_TEXTURE_2D indicates that we are using a 2Dimensional texture, 0 is the mip map level we want to copy the buffer to, the default level is 0, GL_LUMINANCE indicates the format of the data to be copied. I used GL_LUMINANCE because the final result looks better, this way the luminance part of the buffer will be copied to the texture. Other parameters could be GL_ALPHA, GL_RGB, GL_INTENSITY and more.

The next 2 parameters tell OpenGL where to start copying from (0,0). The width and height (128,128) is how many pixels to copy from left to right and how many to copy up and down. The last parameter is only used if we want a border which we dont.

Now that we have a copy of the colour buffer (with the stretched spring) in our BlurTexture we can clear the buffer and set the viewport back to the proper dimensions (640x480 - fullscreen).

IMPORTANT:

This trick can be used only with double buffered pixel formats. The reason why is because all these operations are hidden from the viewer (done on the back buffer).

```
void RenderToTexture() // Renders To A Texture
{
glViewport(0,0,128,128); // Set Our Viewport (Match Texture Size)

ProcessHelix(); // Render The Helix

glBindTexture(GL_TEXTURE_2D,BlurTexture); // Bind To The Blur Texture

// Copy Our ViewPort To The Blur Texture (From 0,0 To 128,128... No Border)
glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, 0, 0, 128, 128, 0);

glClearColor(0.0f, 0.0f, 0.5f, 0.5); // Set The Clear Color To Medium Blue
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And Depth Buffer

glViewport(0 , 0,640 ,480); // Set Viewport (0,0 to 640x480)
}
```

The DrawBlur function simply draws some blended quads in front of our 3D scene, using the BlurTexture we got before. This way, playing a bit with alpha and scaling the texture, we get something that really looks like radial blur.

I first disable GEN_S and GEN_T (I'm addicted to sphere mapping, so my routines usually enable these instructions :P).

We enable 2D texturing, disable depth testing, set the proper blend function, enable blending and then bind the BlurTexture.

The next thing we do is switch to an ortho view, that way it's easier to draw a quad that perfectly fits the screen size. This is how line up the texture over top of the 3D object (by stretching the texture to match the screen ratio). This is where problem two is resolved!

```
void DrawBlur(int times, float inc) // Draw The Blurred Image
{
float spost = 0.0f; // Starting Texture Coordinate Offset
float alphainc = 0.9f / times; // Fade Speed For Alpha Blending
float alpha = 0.2f; // Starting Alpha Value

// Disable AutoTexture Coordinates
glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);

glEnable(GL_TEXTURE_2D); // Enable 2D Texture Mapping
glDisable(GL_DEPTH_TEST); // Disable Depth Testing
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // Set Blending Mode
glEnable(GL_BLEND); // Enable Blending
glBindTexture(GL_TEXTURE_2D,BlurTexture); // Bind To The Blur Texture
ViewOrtho(); // Switch To An Ortho View

alphainc = alpha / times; // alphainc=0.2f / Times To Render Blur
```

We draw the texture many times to create the radial effect, scaling the texture coordinates and raising the blend factor every time we do another pass. We draw 25 quads stretching the texture by 0.015f each time.

```
glBegin(GL_QUADS); // Begin Drawing Quads
for (int num = 0;num < times;num++) // Number Of Times To Render Blur
{
glColor4f(1.0f, 1.0f, 1.0f, alpha); // Set The Alpha Value (Starts At 0.2)
glTexCoord2f(0+spost,1-spost); // Texture Coordinate ( 0, 1 )
glVertex2f(0,0); // First Vertex ( 0, 0 )

glTexCoord2f(0+spost,0+spost); // Texture Coordinate ( 0, 0 )
glVertex2f(0,480); // Second Vertex ( 0, 480 )

glTexCoord2f(1-spost,0+spost); // Texture Coordinate ( 1, 0 )
glVertex2f(640,480); // Third Vertex ( 640, 480 )

glTexCoord2f(1-spost,1-spost); // Texture Coordinate ( 1, 1 )
glVertex2f(640,0); // Fourth Vertex ( 640, 0 )

spost += inc; // Gradually Increase spost (Zooming Closer To Texture Center)
alpha = alpha - alphainc; // Gradually Decrease alpha (Gradually Fading Image Out)
}
glEnd(); // Done Drawing Quads

ViewPerspective(); // Switch To A Perspective View

glEnable(GL_DEPTH_TEST); // Enable Depth Testing
glDisable(GL_TEXTURE_2D); // Disable 2D Texture Mapping
glDisable(GL_BLEND); // Disable Blending
glBindTexture(GL_TEXTURE_2D,0); // Unbind The Blur Texture
}
```

And voila', this is the shortest Draw routine ever seen, with a great looking effect!

We call the RenderToTexture function. This renders the stretched spring once thanks to our viewport change. The stretched spring is rendered to our texture, and the buffers are cleared.

We then draw the "REAL" spring (the 3D object you see on the screen) by calling ProcessHelix( ).

Finally, we draw some blended quads in front of the spring. The textured quads will be stretched to fit over top of the REAL 3D spring.

```
void Draw (void) // Draw The Scene
{
glClearColor(0.0f, 0.0f, 0.0f, 0.5); // Set The Clear Color To Black
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer
glLoadIdentity(); // Reset The View
RenderToTexture(); // Render To A Texture
ProcessHelix(); // Draw Our Helix
DrawBlur(25,0.02f); // Draw The Blur Effect
glFlush (); // Flush The GL Rendering Pipeline
}
```

I hope you enjoyed this tutorial, it really doesn't teach much other than rendering to a texture, but it's definitely an interesting effect to add to your 3d programs.

If you have any comments suggestions or if you know of a better way to implement this effect contact me rio@spinningkids.org.

You are free to use this code however you want in productions of your own, but before you RIP it, give it a look and try to understand what it does, that's the only way ripping is allowed! Also, if you use this code, please, give me some credit!

I want also leave you all with a list of things to do (homework) :D

1) Modify the DrawBlur routine to get an horizontal blur, vertical blur and some more good effects (Twirl blur!).
2) Play with the DrawBlur parameter (add, remove) to get a good routine to sync with your music.
3) Play around with DrawBlur params and a SMALL texture using GL_LUMINANCE (Funky Shininess!).
4) Try superfake volumetric shadows using dark textures instead of luminance one!

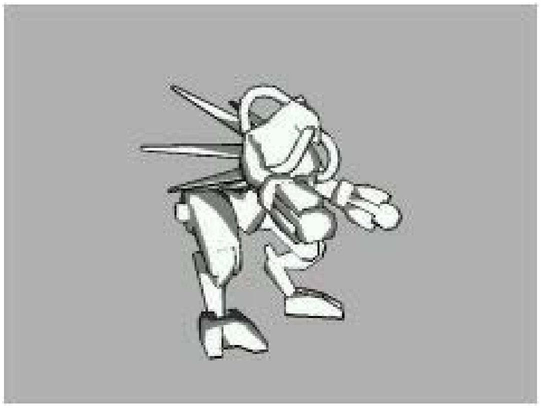
Ok, that should be all for now.

Visit my site and (SK one) for more upcoming tutorials at http://www.spinningkids.org/rio.

Dario Corno (**rIo**)

Jeff Molofee (**NeHe**)

Lesson 37
Cel-Shading



Cel-Shading By Sami "MENTAL" Hamlaoui

Seeing as people still e-mail me asking for source code to the article I wrote on GameDev.net a while ago, and seeing as the 2nd version of that article (with source for every API out there) isn't even close to being halfway finished, I've hacked together this tutorial for NeHe (that was actually going to be the original intention of the article) so all of you OpenGL gurus can play around with it. Sorry for the choice of model, but I've been playing Quake 2 extensivly recently... :)

Note: The original article for this code can be found at: http://www.gamedev.net/reference/programming/features/celshading.

This tutorial doesn't actually explain the theory, just the code. WHY it works can be found at the above link. Now for crying out loud STOP E-MAILING ME REQUESTS FOR SOURCE CODE!!!!

Enjoy :).

First of all, we need to include a few extra header files. The first one (math.h) is so we can use the sqrtf (square root) function, and the second (stdio.h) is for file access.

```
#include <math.h> // Header File For The Math Library
#include <stdio.h> // Header File For The Standard I/O Library
```

Now we are going to define a few structures to help store our data (saves having hundreds of arrays of floats). The first one is the tagMATRIX structure. If you look closely, you will see that we are storing the matrix as a 1D array of 16 floats as opposed to a 2D 4x4 array. This is down to how OpenGL stores it's matrices. If we used 4x4, the values would come out in the wrong order.

```
typedef struct tagMATRIX // A Structure To Hold An OpenGL Matrix
{
float Data[16]; // We Use [16] Due To OpenGL's Matrix Format
}
MATRIX;
```

Second up is the vector class. This simply stores a value for X, Y and Z.

```
typedef struct tagVECTOR // A Structure To Hold A Single Vector
{
float X, Y, Z; // The Components Of The Vector
}
VECTOR;
```

Third, we have the vertex structure. Each vertex only needs it's normal and position (no texture co-ordinates). They MUST be stored in this order, or else when it comes to loading the file things will go horribly wrong (I found out the hard way :(. That'll teach me to hack my code to pieces.).

```
typedef struct tagVERTEX // A Structure To Hold A Single Vertex
{
VECTOR Nor; // Vertex Normal
VECTOR Pos; // Vertex Position
}
VERTEX;
```

Finally, the polygon structure. I know this is a stupid way of storing vertexes, but for the sake of simplicity it works perfectly. Usually I would use an array of vertexes, an array of polygons, and contain the Index number of the 3 verts in the polygon structure, but this is easier to show you what's going on.

```
typedef struct tagPOLYGON // A Structure To Hold A Single Polygon
{
VERTEX Verts[3]; // Array Of 3 VERTEX Structures
}
POLYGON;
```

Pretty simple stuff here too. Look at the comments for an explaination of each variable.

```
bool outlineDraw = true; // Flag To Draw The Outline
bool outlineSmooth = false; // Flag To Anti-Alias The Lines
float outlineColor[3] = { 0.0f, 0.0f, 0.0f }; // Color Of The Lines
float outlineWidth = 3.0f; // Width Of The Lines

VECTOR lightAngle; // The Direction Of The Light
bool lightRotate = false; // Flag To See If We Rotate The Light

float modelAngle = 0.0f; // Y-Axis Angle Of The Model
bool modelRotate = false; // Flag To Rotate The Model

POLYGON *polyData = NULL; // Polygon Data
int polyNum = 0; // Number Of Polygons

GLuint shaderTexture[1]; // Storage For One Texture
```

This is as simple as model file formats get. The first few bytes store the number of polygons in the scene, and the rest of the file is an array of tagPOLYGON structures. Because of this, the data can be read in without any need to sort it into any particular order.

```
BOOL ReadMesh () // Reads The Contents Of The "model.txt" File
{
FILE *In = fopen ("Data\\model.txt", "rb"); // Open The File

if (!In)
return FALSE; // Return FALSE If File Not Opened

fread (&polyNum, sizeof (int), 1, In); // Read The Header (i.e. Number Of Polygons)

polyData = new POLYGON [polyNum]; // Allocate The Memory

fread (&polyData[0], sizeof (POLYGON) * polyNum, 1, In);// Read In All Polygon Data

fclose (In); // Close The File

return TRUE; // It Worked
}
```

Some basic math functions now. The DotProduct calculates the angle between 2 vectors or planes, the Magnitude function calculates the length of the vector, and the Normalize function reduces the vector to a unit length of 1.

```
inline float DotProduct (VECTOR &V1, VECTOR &V2) // Calculate The Angle Between The 2 Vectors
{
return V1.X * V2.X + V1.Y * V2.Y + V1.Z * V2.Z; // Return The Angle
}

inline float Magnitude (VECTOR &V) // Calculate The Length Of The Vector
{
return sqrtf (V.X * V.X + V.Y * V.Y + V.Z * V.Z); // Return The Length Of The Vector
}

void Normalize (VECTOR &V) // Creates A Vector With A Unit Length Of 1
{
float M = Magnitude (V); // Calculate The Length Of The Vector

if (M != 0.0f) // Make Sure We Don't Divide By 0
{
V.X /= M; // Normalize The 3 Components
V.Y /= M;
V.Z /= M;
}
}
```

This function rotates a vector using the matrix provided. Please note that it ONLY rotates the vector - it has nothing to do with the position of the vector. This is used when rotating normals to make sure that they stay pointing in the right direction when we calculate the lighting.

```
void RotateVector (MATRIX &M, VECTOR &V, VECTOR &D) // Rotate A Vector Using The Supplied Matrix
{
D.X = (M.Data[0] * V.X) + (M.Data[4] * V.Y) + (M.Data[8] * V.Z); // Rotate Around The X Axis
D.Y = (M.Data[1] * V.X) + (M.Data[5] * V.Y) + (M.Data[9] * V.Z); // Rotate Around The Y Axis
D.Z = (M.Data[2] * V.X) + (M.Data[6] * V.Y) + (M.Data[10] * V.Z); // Rotate Around The Z Axis
}
```

The first major function of the engine, Initialize does exactly what is says. I've cut out a few lines of code as they are not needed in the explaination.

```
// Any GL Init Code & User Initialiazation Goes Here
BOOL Initialize (GL_Window* window, Keys* keys)
{
```

These 3 variables are used to load the shader file. Line contains space for a single line in the text file, while shaderData stores the actual shader values. You may be wondering why we have 96 values instead of 32. Well, we need to convert the greyscale values to RGB so that OpenGL can use them. We can still store the values as greyscale, but we will simply use the same value for the R, G and B components when uploading the texture.

```
char Line[255]; // Storage For 255 Characters
float shaderData[32][3]; // Storate For The 96 Shader Values

FILE *In = NULL; // File Pointer
```

When drawing the lines, we want to make sure that they are nice and smooth. Initially this value is turned off, but by pressing the "2" key, it can be toggled on/off.

```
glShadeModel (GL_SMOOTH); // Enables Smooth Color Shading
glDisable (GL_LINE_SMOOTH); // Initially Disable Line Smoothing

glEnable (GL_CULL_FACE); // Enable OpenGL Face Culling
```

We disable OpenGL lighting because we do all of the lighting calculations ourself.

```
glDisable (GL_LIGHTING); // Disable OpenGL Lighting
```

Here is where we load the shader file. It is simply 32 floating point values stored as ASCII (for easy modification), each one on a seperate line.

```
In = fopen ("Data\\shader.txt", "r"); // Open The Shader File

if (In) // Check To See If The File Opened
{
for (i = 0; i < 32; i++) // Loop Though The 32 Greyscale Values
{
if (feof (In)) // Check For The End Of The File
break;

fgets (Line, 255, In); // Get The Current Line
```

Here we convert the greyscale value into RGB, as described above.

```
// Copy Over The Value
shaderData[i][0] = shaderData[i][1] = shaderData[i][2] = atof (Line);
}

fclose (In); // Close The File
}

else
return FALSE; // It Went Horribly Horribly Wrong
```

Now we upload the texture. At it clearly states, do not use any kind of filtering on the texture or else it will look odd, to say the least. GL_TEXTURE_1D is used because it is a 1D array of values.

```
glGenTextures (1, &shaderTexture[0]); // Get A Free Texture ID

glBindTexture (GL_TEXTURE_1D, shaderTexture[0]); // Bind This Texture. From Now On It Will Be 1D

// For Crying Out Loud Don't Let OpenGL Use Bi/Trilinear Filtering!
glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// Upload
glTexImage1D (GL_TEXTURE_1D, 0, GL_RGB, 32, 0, GL_RGB , GL_FLOAT, shaderData);
```

Now set the lighting direction. I've got it pointing down positive Z, which means it's going to hit the model face-on.

```
lightAngle.X = 0.0f; // Set The X Direction
lightAngle.Y = 0.0f; // Set The Y Direction
lightAngle.Z = 1.0f; // Set The Z Direction

Normalize (lightAngle); // Normalize The Light Direction
```

Load in the mesh from file (described above).

```
return ReadMesh (); // Return The Value Of ReadMesh
}
```

The opposite of the above function, Deinitalize deletes the texture and polygon data created by Initalize and ReadMesh.

```
void Deinitialize (void) // Any User DeInitialization Goes Here
{
```

```
glDeleteTextures (1, &shaderTexture[0]); // Delete The Shader Texture

delete [] polyData; // Delete The Polygon Data
}
```

The main demo loop. All this does is process the input and update the angle. Controls are as follows:

<SPACE> = Toggle rotation

1 = Toggle outline drawing

2 = Toggle outline anti-alaising

<UP> = Increase line width

<DOWM> = Decrease line width

```
void Update (DWORD milliseconds) // Perform Motion Updates Here
{
if (g_keys->keyDown [' '] == TRUE) // Is the Space Bar Being Pressed?
{
modelRotate = !modelRotate; // Toggle Model Rotation On/Off

g_keys->keyDown [' '] = FALSE;
}

if (g_keys->keyDown ['1'] == TRUE) // Is The Number 1 Being Pressed?
{
outlineDraw = !outlineDraw; // Toggle Outline Drawing On/Off

g_keys->keyDown ['1'] = FALSE;
}

if (g_keys->keyDown ['2'] == TRUE) // Is The Number 2 Being Pressed?
{
outlineSmooth = !outlineSmooth; // Toggle Anti-Aliasing On/Off

g_keys->keyDown ['2'] = FALSE;
}

if (g_keys->keyDown [VK_UP] == TRUE) // Is The Up Arrow Being Pressed?
{
outlineWidth++; // Increase Line Width

g_keys->keyDown [VK_UP] = FALSE;
}

if (g_keys->keyDown [VK_DOWN] == TRUE) // Is The Down Arrow Being Pressed?
{
outlineWidth--; // Decrease Line Width

g_keys->keyDown [VK_DOWN] = FALSE;
}

if (modelRotate) // Check To See If Rotation Is Enabled
modelAngle += (float) (milliseconds) / 10.0f; // Update Angle Based On The Clock
}
```

The function you've all been waiting for. The Draw function does everything - calculates the shade values, renders the mesh, renders the outline, and, well that's it really.

```
void Draw (void)
{
```

TmpShade is used to store the shader value for the current vertex. All vertex data is calculate at the same time, meaning that we only need to use a single variable that we can just keep reusing.

The TmpMatrix, TmpVector and TmpNormal structures are also used to calculate the vertex data. TmpMatrix is set once at the start of the function and never changed until Draw is called again. TmpVector and TmpNormal on the other hand, change when another vertex is processed.

```
float TmpShade; // Temporary Shader Value

MATRIX TmpMatrix; // Temporary MATRIX Structure
VECTOR TmpVector, TmpNormal; // Temporary VECTOR Structures
```

Lets clear the buffers and matrix data.

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Buffers
glLoadIdentity (); // Reset The Matrix
```

The first check is to see if we want to have smooth outlines. If so, then we turn on anti-alaising. If not, we turn it off. Simple!

```
if (outlineSmooth) // Check To See If We Want Anti-Aliased Lines
```

```
{
glHint (GL_LINE_SMOOTH_HINT, GL_NICEST); // Use The Good Calculations
glEnable (GL_LINE_SMOOTH); // Enable Anti-Aliasing
}

else // We Don't Want Smooth Lines
glDisable (GL_LINE_SMOOTH); // Disable Anti-Aliasing
```

We then setup the viewport. We move the camera back 2 units, and then rotate the model by the angle. Note: because we moved the camera first, the model will rotate on the spot. If we did it the other way around, the model would rotate around the camera.

We then grab the newly created matrix from OpenGL and store it in TmpMatrix.

```
glTranslatef (0.0f, 0.0f, -2.0f); // Move 2 Units Away From The Screen
glRotatef (modelAngle, 0.0f, 1.0f, 0.0f); // Rotate The Model On It's Y-Axis

glGetFloatv (GL_MODELVIEW_MATRIX, TmpMatrix.Data); // Get The Generated Matrix
```

The magic begins. We first enable 1D texturing, and then enable the shader texture. This is to be used as a look-up table by OpenGL. We then set the color of the model (white). I chose white because it shows up the highlights and shading much better then other colors. I suggest that you don't use black :)

```
// Cel-Shading Code
glEnable (GL_TEXTURE_1D); // Enable 1D Texturing
glBindTexture (GL_TEXTURE_1D, shaderTexture[0]); // Bind Our Texture

glColor3f (1.0f, 1.0f, 1.0f); // Set The Color Of The Model
```

Now we start drawing the triangles. We look though each polygon in the array, and then in turn each of it's vertexes. The first step is to copy the normal information into a temporary structure. This is so we can rotate the normals, but still keep the original values preserved (no precision degradation).

```
glBegin (GL_TRIANGLES); // Tell OpenGL That We're Drawing Triangles

for (i = 0; i < polyNum; i++) // Loop Through Each Polygon
{
for (j = 0; j < 3; j++) // Loop Through Each Vertex
{
TmpNormal.X = polyData[i].Verts[j].Nor.X; // Fill Up The TmpNormal Structure With The
TmpNormal.Y = polyData[i].Verts[j].Nor.Y; // Current Vertices' Normal Values
TmpNormal.Z = polyData[i].Verts[j].Nor.Z;
```

Second, we rotate the normal by the matrix grabbed from OpenGL earlier. We then normalize this so it doesn't go all screwy.

```
// Rotate This By The Matrix
RotateVector (TmpMatrix, TmpNormal, TmpVector);

Normalize (TmpVector); // Normalize The New Normal
```

Third, we get the dot product of the rotated normal and light direction (called lightAngle, because I forgot to change it from my old light class). We then clamp the value to the range 0-1 (from -1 to +1).

```
// Calculate The Shade Value
TmpShade = DotProduct (TmpVector, lightAngle);

if (TmpShade < 0.0f)
TmpShade = 0.0f; // Clamp The Value to 0 If Negative
```

Forth, we pass this value to OpenGL as the texture co-ordinate. The shader texture acts as a lookup table (the shader value being the index), which is (i think) the main reason why 1D textures were invented. We then pass the vertex's position to OpenGL, and repeat. And Repeat. And Repeat. And I think you get the idea.

```
glTexCoord1f (TmpShade); // Set The Texture Co-ordinate As The Shade Value
// Send The Vertex Position
glVertex3fv (&polyData[i].Verts[j].Pos.X);
}
}

glEnd (); // Tell OpenGL To Finish Drawing

glDisable (GL_TEXTURE_1D); // Disable 1D Textures
```

Now we move onto the outlines. An outline can be defined as "an edge where one polygon is front facing, and the other is backfacing". In OpenGL, it's where the depth test is set to less than or equal to (GL_LEQUAL) the current value, and when all front faces are being culled. We also blend the lines in, to make it look nice :)

So, we enable blending and set the blend mode. We tell OpenGL to render backfacing polygons as lines, and set the width of those lines. We cull all front facing polygons, and set the depth test to less than or equal to the current Z value. After this the color of the line is set, and we loop though each polygon, drawing its vertices. We only need to pass the vertex position, and not the normal or shade value because all we want is an outline.

```
// Outline Code
```

```
if (outlineDraw) // Check To See If We Want To Draw The Outline
{
glEnable (GL_BLEND); // Enable Blending
// Set The Blend Mode
glBlendFunc (GL_SRC_ALPHA ,GL_ONE_MINUS_SRC_ALPHA);

glPolygonMode (GL_BACK, GL_LINE); // Draw Backfacing Polygons As Wireframes
glLineWidth (outlineWidth); // Set The Line Width

glCullFace (GL_FRONT); // Don't Draw Any Front-Facing Polygons

glDepthFunc (GL_LEQUAL); // Change The Depth Mode

glColor3fv (&outlineColor[0]); // Set The Outline Color

glBegin (GL_TRIANGLES); // Tell OpenGL What We Want To Draw

for (i = 0; i < polyNum; i++) // Loop Through Each Polygon
{
for (j = 0; j < 3; j++) // Loop Through Each Vertex
{
// Send The Vertex Position
glVertex3fv (&polyData[i].Verts[j].Pos.X);
}
}

glEnd (); // Tell OpenGL We've Finished
```

After this, we just set everything back to how it was before, and exit.

```
glDepthFunc (GL_LESS); // Reset The Depth-Testing Mode

glCullFace (GL_BACK); // Reset The Face To Be Culled

glPolygonMode (GL_BACK, GL_FILL); // Reset Back-Facing Polygon Drawing Mode

glDisable (GL_BLEND); // Disable Blending
}
}
```

Sami Hamlaoui (**MENTAL**)

Jeff Molofee (**NeHe**)

Lesson 38 Loading Textures From A Resource File & Texturing Triangles

Welcome to the 38th NeHe Productions Tutorial. It's been awhile since my last tutorial, so my writing may be a little rusty. That and the fact that I've been up for almost 24 hours working on the code :)

So you know how to texture map a quad, and you know how to load bitmap images, tga's, etc. So how the heck do you texture map a Triangle? And what if you want to hide your textures in the .EXE file?

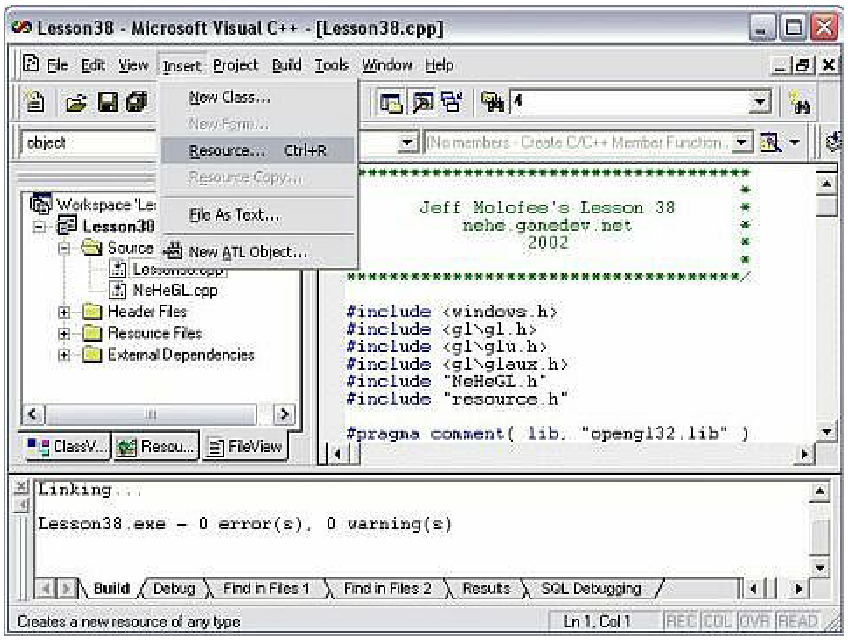
The two questions I'm asked on a daily basis will soon be answered, and once you see how easy it is, you'll wonder why you never thought of the solution :)

Rather than trying to explain everything in great detail I'm going to include a few screenshots, so you know exactly what it is I'm talking about. I will be using the latest basecode. You can download the code from the main page under the heading "NeHeGL I Basecode" or you can download the code at the end of this tutorial.

The first thing we need to do is add the images to the resource file. Alot of you have already figured out how to do this, unfortunately, you miss a few steps along the way and end up with a useless resource file filled with bitmaps that you can't use.

Remember, this tutorial was written in Visual C++ 6.0. If you're using anything other than Visual C++, the resource portion of this tutorial wont make sense (especially the screenshots).

* Currently you can only use 24 bit BMP images. There is alot of extra code to load 8 bit BMP files. I'd love to hear from anyone that has a tiny / optimized BMP loader. The code I have right now to load 8 and 24 bit BMP's is a mess. Something that uses LoadImage would be nice.

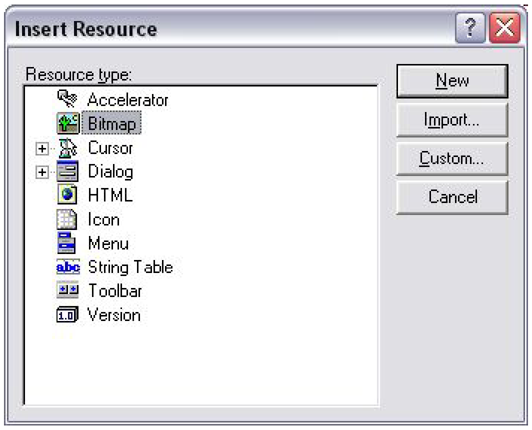

Lesson38 - Microsoft Visual C++ - [Lesson38.cpp]
File Edit View Insert Project Build Tools Window Help
New Class...
New Form...
Resource... Ctrl+R
Resource Copy...
File As Text...
New ATL Object...
object
(No members - Create C/C++ Member Function...)
Workspace 'Le
Lesson38
Source
NeHeGL.cpp
Header Files
Resource Files
External Dependencies
ClassV... Resou... FileView
Build Debug Find in Files 1 Find in Files 2 Results SQL Debugging
Ln 1, Col 1 REC COL OVR READ

```
*****************************************
      Jeff Molofee's Lesson 38           *
         nehe.gamedev.net                *
               2002                      *
                                         *
****************************************/

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include "NeHeGL.h"
#include "resource.h"

#pragma comment( lib, "opengl32.lib" )
```

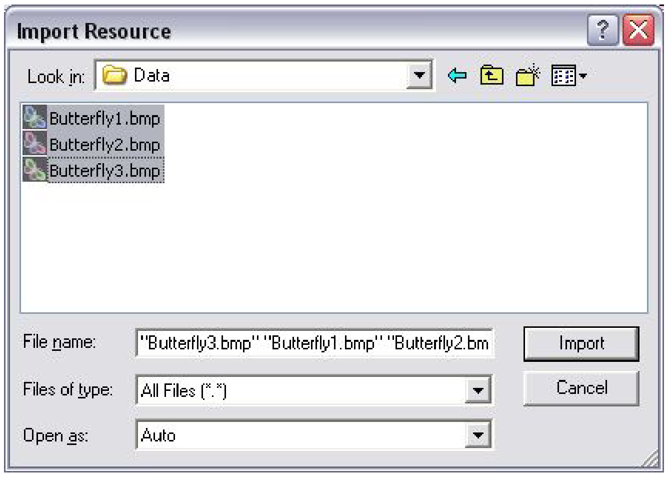
```
Linking...

Lesson38.exe - 0 error(s), 0 warning(s)
```

Creates a new resource of any type

Open the project and click INSERT on the main menu. Once the INSERT menu has opened, select RESOURCE.



You are now asked what type of resource you wish to import. Select BITMAP and click the IMPORT button.

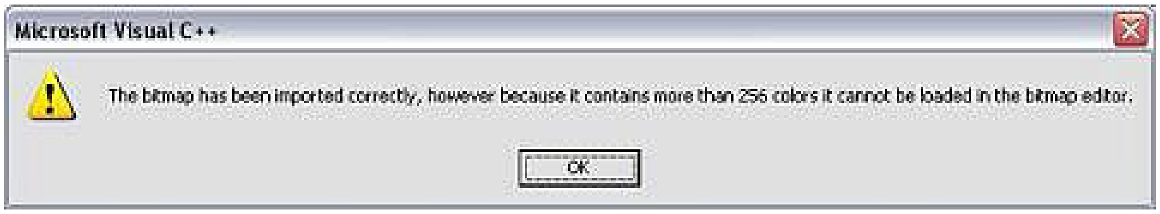


File name: "Butterfly3.bmp" "Butterfly1.bmp" "Butterfly2.bm

Files of type: All Files (*.*)

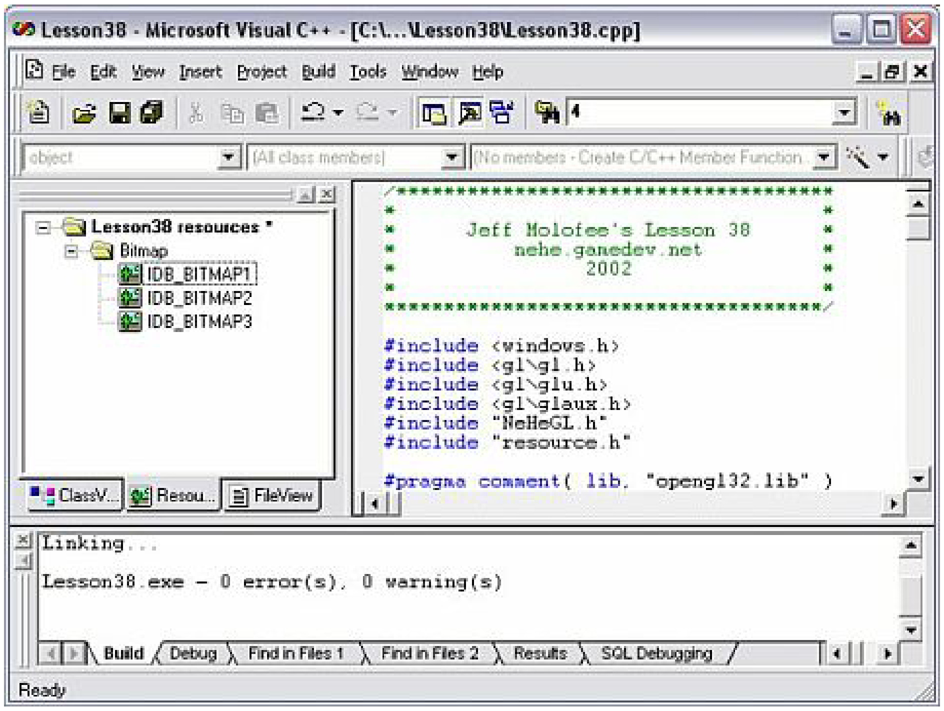
Open as: Auto

A file selection box will open. Browse to the DATA directory, and highlight all three images (Hold down the CTRL key while selecting each image). Once you have all three selected click the IMPORT button. If You do not see the bitmap files, make sure FILES OF TYPE at the bottom says ALL FILE (*.*).


Microsoft Visual C++

The bitmap has been imported correctly, however because it contains more than 256 colors it cannot be loaded in the bitmap editor.

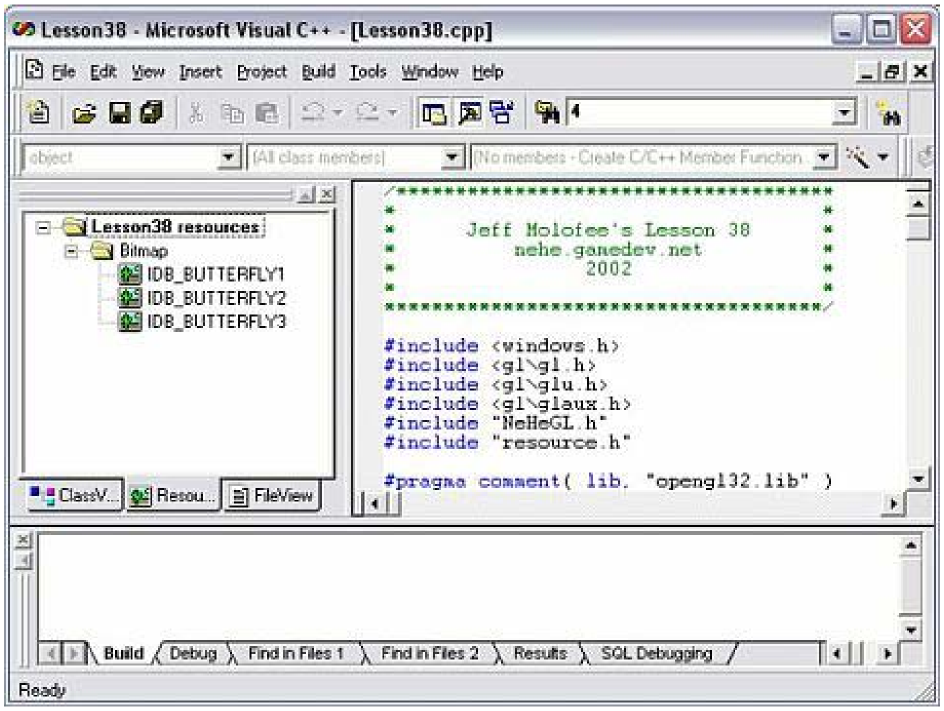
A warning will pop up three times (once for each image you imported). All it's telling you is that the image was imported fine, but the picture can't be viewed or edited because it has more than 256 colors. Nothing to worry about!


Lesson38 - Microsoft Visual C++ - [C:\...\Lesson38\Lesson38.cpp]

```
/****************************************
*                                       *
*       Jeff Molofee's Lesson 38        *
*          nehe.gamedev.net             *
*               2002                    *
*                                       *
****************************************/

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include "NeHeGL.h"
#include "resource.h"

#pragma comment( lib, "opengl32.lib" )
```

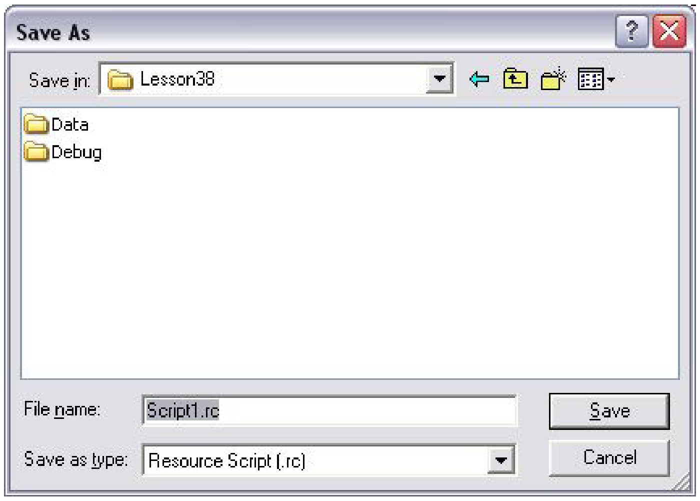
```
Linking...

Lesson38.exe - 0 error(s), 0 warning(s)
```

Once all three images have been imported, a list will be displayed. Each bitmap has been assigned an ID. Each ID starts with IDB_BITMAP and then a number from 1 - 3. If you were lazy, you could leave the ID's and jump to the code. Lucky we're not lazy!


Lesson38 - Microsoft Visual C++ - [Lesson38.cpp]
File Edit View Insert Project Build Tools Window Help
Lesson38 resources
Bitmap
IDB_BUTTERFLY1
IDB_BUTTERFLY2
IDB_BUTTERFLY3
ClassV... Resou... FileView
Build Debug Find in Files 1 Find in Files 2 Results SQL Debugging
Ready

```
/****************************************
*                                       *
*     Jeff Molofee's Lesson 38          *
*         nehe.gamedev.net              *
*              2002                     *
*                                       *
****************************************/

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include "NeHeGL.h"
#include "resource.h"

#pragma comment( lib, "opengl32.lib" )
```

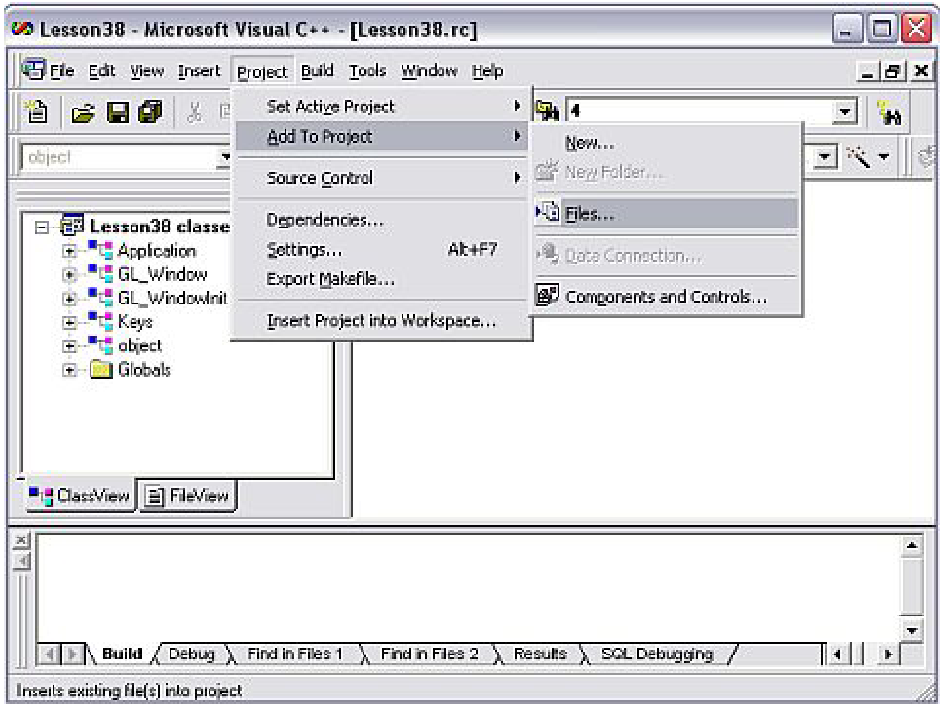
Right click each ID, and select PROPERTIES. Rename each ID so that it matches the name of the original bitmap file. See the picture if you're not sure what I mean.



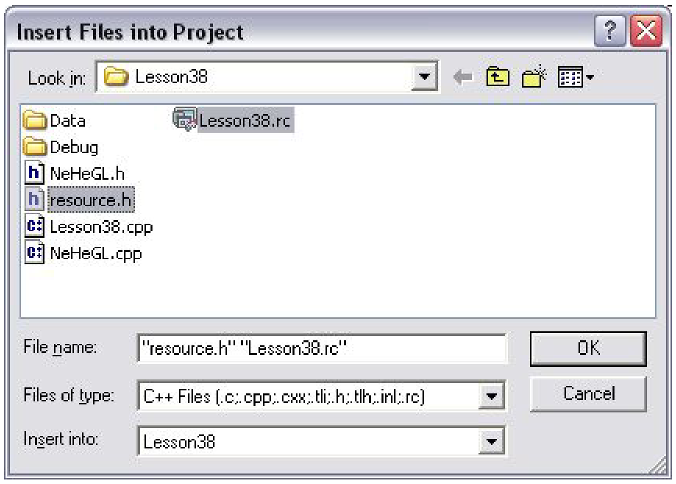
File name: Script1.rc
Save as type: Resource Script (.rc)

Once you are done, select FILE from the main menu and then SAVE ALL because you have just created a new resource file, windows will ask you what you want to call the file. You can save the file with the default filename or you can rename it to lesson38.rc. Once you have decided on a name click SAVE.

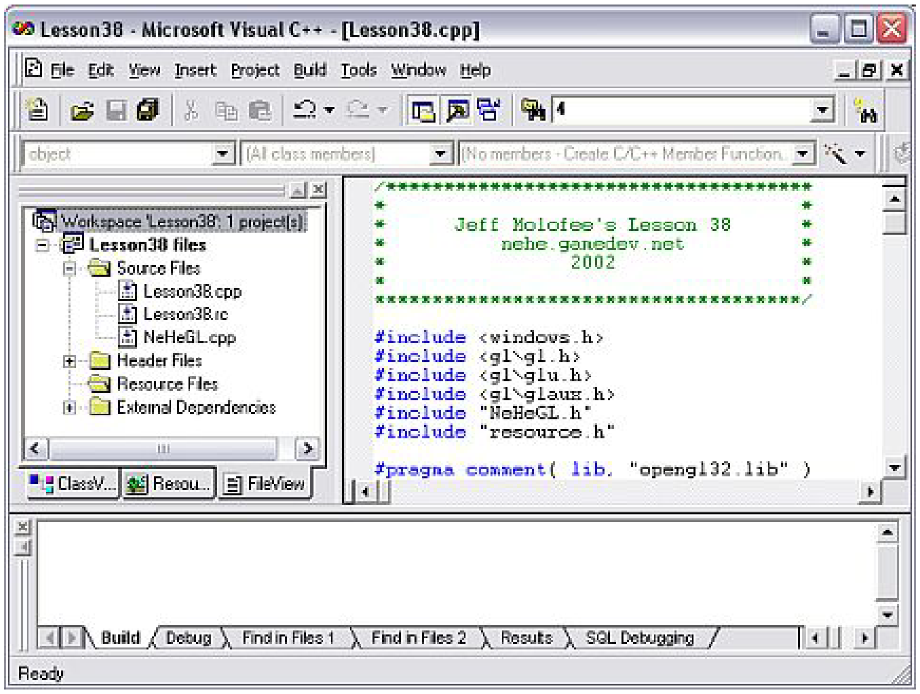
This is the point that most people make it to. You have a resource file. It's full of Bitmap images and it's been saved to the Hard Drive. To use the images, you need to complete a few more steps.



The next thing you need to do is add the resource file to your current project. Select PROJECT from the main menu, ADD TO PROJECT, and then FILES.


Insert Files into Project
Look in: Lesson38
Data
Debug
NeHeGL.h
resource.h
Lesson38.cpp
NeHeGL.cpp
Lesson38.rc
File name: "resource.h" "Lesson38.rc"
Files of type: C++ Files (.c;.cpp;.cxx;.tli;.h;.tlh;.inl;.rc)
Insert into: Lesson38
OK
Cancel

Select the resource.h file, and the resource file (Lesson38.rc). Hold down control to select more than one file, or add each file individually.


Lesson38 - Microsoft Visual C++ - [Lesson38.cpp]
File Edit View Insert Project Build Tools Window Help
Workspace 'Lesson38': 1 project(s)
Lesson38 files
Source Files
Lesson38.cpp
Lesson38.rc
NeHeGL.cpp
Header Files
Resource Files
External Dependencies
ClassV... Resou... FileView
Build Debug Find in Files 1 Find in Files 2 Results SQL Debugging
Ready

```
/****************************************
*                                       *
*      Jeff Molofee's Lesson 38         *
*         nehe.gamedev.net              *
*              2002                     *
*                                       *
****************************************/

#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include "NeHeGL.h"
#include "resource.h"

#pragma comment( lib, "opengl32.lib" )
```

The last thing to do is make sure the resource file (Lesson38.rc) was put in the RESOURCE FILES folder. As you can see in the picture above, it was put in the SOURCE FILES folder. Click it with your mouse and drag it down to the RESOURCE FILES folder.

Once the resource file has been moved select FILE from the main menu and SAVE ALL. The hard part has been done! ...Way too many pictures :)

So now we start on the code! The most important line in the section of code below is #include "resource.h". Without this line, you will get a bunch of undeclared identifier errors when you try to compile the code. The resource.h file declares the objects inside the resource file. So if you want to grab data from IDB_BUTTERFLY1 you had better remember to include the header file!

```
#include <windows.h> // Header File For Windows
#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
#include "NeHeGL.h" // Header File For NeHeGL
#include "resource.h" // Header File For Resource (*IMPORTANT*)

#pragma comment( lib, "opengl32.lib" ) // Search For OpenGL32.lib While Linking
#pragma comment( lib, "glu32.lib" ) // Search For GLu32.lib While Linking
#pragma comment( lib, "glaux.lib" ) // Search For GLaux.lib While Linking

#ifndef CDS_FULLSCREEN // CDS_FULLSCREEN Is Not Defined By Some
#define CDS_FULLSCREEN 4 // Compilers. By Defining It This Way,
#endif // We Can Avoid Errors

GL_Window* g_window;
Keys* g_keys;
```

The first line below sets aside space for the three textures we're going to make.

The structure will be used to hold information about 50 different objects that we'll have moving around the screen.

tex will keep track of which texture to use for the object. x is the xposition of the object, y is the y position of the object, z is the objects position on the z-axis, yi will be a random number used to control how fast the object falls. spinz will be used to rotate the object on it's z-axis as it falls, spinzi is another random number used to control how fast the object spins. flap will be used to control the objects wings (more on this later) and fi is a random value that controls how fast the wings flap.

We create 50 instances of obj[ ] based on the object structure.

```
// User Defined Variables
GLuint texture[3]; // Storage For 3 Textures

struct object // Create A Structure Called Object
{
int tex; // Integer Used To Select Our Texture
float x; // X Position
float y; // Y Position
float z; // Z Position
float yi; // Y Increase Speed (Fall Speed)
float spinz; // Z Axis Spin
float spinzi; // Z Axis Spin Speed
float flap; // Flapping Triangles :)
float fi; // Flap Direction (Increase Value)
};

object obj[50]; // Create 50 Objects Using The Object Structure
```

The bit of code below assigns random startup values to object (obj[ ]) loop. loop can be any value from 0 - 49 (any one of the 50 objects).

We start off with a random texture from 0 to 2. This will select a random colored butterfly.

We assign a random x position from -17.0f to +17.0f. The starting y position will be 18.0f, which will put the object just above the screen so we can't see it right off the start.

The z position is also a random value from -10.0f to -40.0f. The spinzi value is a random value from -1.0f to 1.0f. flap is set to 0.0f (which will be the center position for the wings).

Finally, the flap speed (fi) and fall speed (yi) are also given a random value.

```
void SetObject(int loop) // Sets The Initial Value Of Each Object (Random)
{
obj[loop].tex=rand()%3; // Texture Can Be One Of 3 Textures
obj[loop].x=rand()%34-17.0f; // Random x Value From -17.0f To 17.0f
obj[loop].y=18.0f; // Set y Position To 18 (Off Top Of Screen)
obj[loop].z=-((rand()%30000/1000.0f)+10.0f); // z Is A Random Value From -10.0f To -40.0f
obj[loop].spinzi=(rand()%10000)/5000.0f-1.0f; // spinzi Is A Random Value From -1.0f To 1.0f
obj[loop].flap=0.0f; // flap Starts Off At 0.0f;
obj[loop].fi=0.05f+(rand()%100)/1000.0f; // fi Is A Random Value From 0.05f To 0.15f
obj[loop].yi=0.001f+(rand()%1000)/10000.0f; // yi Is A Random Value From 0.001f To 0.101f
}
```

Now for the fun part! Loading a bitmap from the resource file and converting it to a texture.

hBMP is a pointer to our bitmap file. It will tell our program where to get the data from. BMP is a bitmap structure that we can fill with data from our resource file.

We tell our program which ID's to use in the third line of code. We want to load IDB_BUTTEFLY1, IDB_BUTTEFLY2 and IDB_BUTTERFLY3. If you wish to add more images, add the image to the resource file, and add the ID to Texture[ ].

```
void LoadGLTextures() // Creates Textures From Bitmaps In The Resource File
{
HBITMAP hBMP; // Handle Of The Bitmap
BITMAP BMP; // Bitmap Structure

// The ID Of The 3 Bitmap Images We Want To Load From The Resource File
byte Texture[]={ IDB_BUTTERFLY1, IDB_BUTTERFLY2, IDB_BUTTERFLY3 };
```

The line below uses sizeof(Texture) to figure out how many textures we want to build. We have 3 ID's in Texture[ ] so the value will be 3. sizeof(Texture) is also used for the main loop.

```
glGenTextures(sizeof(Texture), &texture[0]); // Generate 3 Textures (sizeof(Texture)=3 ID's)
for (int loop=0; loop<sizeof(Texture); loop++) // Loop Through All The ID's (Bitmap Images)
{
```

LoadImage takes the following parameters: GetModuleHandle(NULL) - A handle to an instance. MAKEINTRESOURCE(Texture[loop]) - Converts an Integer Value (Texture[loop]) to a resource value (this is the image to load). IMAGE_BITMAP - Tells our program that the resource to load is a bitmap image.

The next two parameters (0,0) are the desired height and width of the image in pixels. We want to use the default size so we set both to 0.

The last parameter (LR_CREATEDIBSECTION) returns a DIB section bitmap, which is a bitmap without all the color information stored in the data. Exactly what we need.

hBMP points to the bitmap data that is loaded by LoadImage( ).

```
hBMP=(HBITMAP)LoadImage(GetModuleHandle(NULL),MAKEINTRESOURCE(Texture[loop]), IMAGE_BITMAP, 0,
0, LR_CREATEDIBSECTION);
```

Next we check to see if the pointer (hBMP) actually points to data. If you wanted to add error checking, you could check hBMP and pop up a messagebox if there's no data.

If data exists, we use GetObject( ) to grab all of the data (sizeof(BMP)) from hBMP and store it in our BMP (bitmap structure).

glPixelStorei tells OpenGL that the data is stored in word alignments (4 bytes per pixel).

We then bind to our texture, set the filtering to GL_LINEAR_MIPMAP_LINEAR (nice and smooth), and generate the texture.

Notice that we use BMP.bmWidth and BMP.bmHeight to get the height and width of the bitmap. We also have to swap the Red and Blue colors using GL_BGR_EXT. The actual resource data is retreived from BMP.bmBits.

The last step is to delete the bitmap object freeing all system resources associated with the object.

```
if (hBMP) // Does The Bitmap Exist?
{ // If So...
GetObject(hBMP,sizeof(BMP), &BMP); // Get The Object
// hBMP: Handle To Graphics Object
// sizeof(BMP): Size Of Buffer For Object Information
// Buffer For Object Information
glPixelStorei(GL_UNPACK_ALIGNMENT,4); // Pixel Storage Mode (Word Alignment / 4 Bytes)
glBindTexture(GL_TEXTURE_2D, texture[loop]); // Bind Our Texture
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR); // Linear Filtering
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEAR); // Mipmap Linear
Filtering

// Generate Mipmapped Texture (3 Bytes, Width, Height And Data From The BMP)
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, BMP.bmWidth, BMP.bmHeight, GL_BGR_EXT, GL_UNSIGNED_BYTE,
BMP.bmBits);
DeleteObject(hBMP); // Delete The Bitmap Object
}
}
}
```

Nothing really fancy in the init code. We add LoadGLTextures() to call the code above. The screen clear color is black. Depth testing is disabled (cheap way to blend). We enable texture mapping, then set up and enable blending.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Any GL Init Code & User Initialiazation Goes
Here
{
g_window = window;
g_keys = keys;

// Start Of User Initialization
LoadGLTextures(); // Load The Textures From Our Resource File

glClearColor (0.0f, 0.0f, 0.0f, 0.5f); // Black Background
glClearDepth (1.0f); // Depth Buffer Setup
glDepthFunc (GL_LEQUAL); // The Type Of Depth Testing (Less Or Equal)
glDisable(GL_DEPTH_TEST); // Disable Depth Testing
glShadeModel (GL_SMOOTH); // Select Smooth Shading
glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Set Perspective Calculations To Most
Accurate
glEnable(GL_TEXTURE_2D); // Enable Texture Mapping
glBlendFunc(GL_ONE,GL_SRC_ALPHA); // Set Blending Mode (Cheap / Quick)
glEnable(GL_BLEND); // Enable Blending
```

We need to initialize all 50 objects right off the start so they don't appear in the middle of the screen or all in the same location. The loop below does just that.

```
for (int loop=0; loop<50; loop++) // Loop To Initialize 50 Objects
{
SetObject(loop); // Call SetObject To Assign New Random Values
}

return TRUE; // Return TRUE (Initialization Successful)
}

void Deinitialize (void) // Any User DeInitialization Goes Here
{
}

void Update (DWORD milliseconds) // Perform Motion Updates Here
{
if (g_keys->keyDown [VK_ESCAPE] == TRUE) // Is ESC Being Pressed?
{
TerminateApplication (g_window); // Terminate The Program
}

if (g_keys->keyDown [VK_F1] == TRUE) // Is F1 Being Pressed?
{
ToggleFullscreen (g_window); // Toggle Fullscreen Mode
}
}
```

Now for the drawing code. In this section I'll attempt to explain the easiest way to texture map a single image across two triangles. For some reason everyone seems to think it's near impossible to texture an image to a triangle.

The truth is, you can texture an image to any shape you want. With very little effort. The image can match the shape or it can be a completely different pattern. It really doesn't matter.

First things first... we clear the screen and set up a loop to render all 50 of our butterflies (objects).

```
void Draw (void) // Draw The Scene
{
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen And Depth Buffer

for (int loop=0; loop<50; loop++) // Loop Of 50 (Draw 50 Objects)
{
```

We call glLoadIdentity( ) to reset the modelview matrix. Then we select the texture that was assigned to our object (obj[loop].tex).

We position the butterfly using glTranslatef() then rotate the buttefly 45 degrees on it's X axis. This tilts the butterfly a little more towards the viewer so it doesn't look like a flat 2D object.

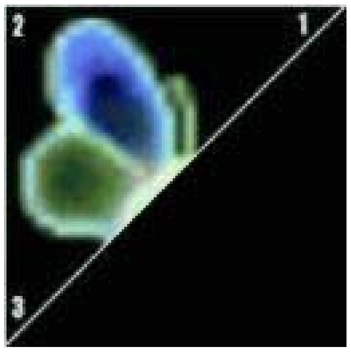
The final rotation spins the butterfly on it's z-axis which makes it spin as it falls down the screen.

```
glLoadIdentity (); // Reset The Modelview Matrix
glBindTexture(GL_TEXTURE_2D, texture[obj[loop].tex]); // Bind Our Texture
glTranslatef(obj[loop].x,obj[loop].y,obj[loop].z); // Position The Object
glRotatef(45.0f,1.0f,0.0f,0.0f); // Rotate On The X-Axis
glRotatef((obj[loop].spinz),0.0f,0.0f,1.0f); // Spin On The Z-Axis
```

Texturing a triangle is not all that different from texturing a quad. Just because you only have 3 vertices doesn't mean you can't texture a quad to your triangle. The only difference is that you need to be more aware of your texture coordinates.

In the code below, we draw the first triangle. We start at the top right corner of an invisible quad. We then move left until we get to the top left corner. From there we go to the bottom left corner.

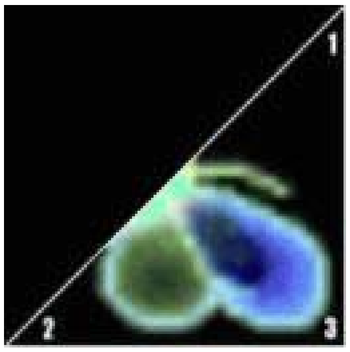
The code below will render the following image:



Notice that half the buttefly is rendered on the first triangle. The other half is rendered on the second triangle. The texture coordinates match up with the vertex coordinates and although there are only 3 texture coordinates, it's still enough information to tell OpenGL what portion of the image needs to be mapped to the triangle.

```
glBegin(GL_TRIANGLES); // Begin Drawing Triangles
// First Triangle
glTexCoord2f(1.0f,1.0f); glVertex3f( 1.0f, 1.0f, 0.0f); // Point 1 (Top Right)
glTexCoord2f(0.0f,1.0f); glVertex3f(-1.0f, 1.0f, obj[loop].flap); // Point 2 (Top Left)
glTexCoord2f(0.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f); // Point 3 (Bottom Left)
```

The code below renders the second half of the triangle. Same technique as above, but this time we render from the top right to the bottom left, then over to the bottom right.



The second point of the first triangle and the third point of the second triangle move back and forth on the z-axis to create the illusion of flapping. What's really happening is that point is moving from -1.0f to 1.0f and then back, which causes the two triangles to bend in the center where the butterflies body is.

If you look at the two pictures you will notice that points 2 and 3 are the tips of the wings. Creates a very nice flapping effect with minimal effort.

```
// Second Triangle
glTexCoord2f(1.0f,1.0f); glVertex3f( 1.0f, 1.0f, 0.0f); // Point 1 (Top Right)
glTexCoord2f(0.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f); // Point 2 (Bottom Left)
glTexCoord2f(1.0f,0.0f); glVertex3f( 1.0f,-1.0f, obj[loop].flap); // Point 3 (Bottom Right)

glEnd(); // Done Drawing Triangles
```

The following bit of code moves the butterfly down the screen by subtracting obj[loop].yi from obj[loop].y. The butterfly spinz value is increased by spinzi (which can be a negative or positive value) and the wings are increased by fi. fi can also be a negative or positive direction depending on the direction we want the wings to flap.

```
obj[loop].y-=obj[loop].yi; // Move Object Down The Screen
obj[loop].spinz+=obj[loop].spinzi; // Increase Z Rotation By spinzi
obj[loop].flap+=obj[loop].fi; // Increase flap Value By fi
```

After moving the butterfly down the screen, we need to see if it's gone past the bottom of the screen (no longer visible). if it has, we call SetObject(loop) to assign the butterfly a new texture, new fall speed, etc.

```
if (obj[loop].y<-18.0f) // Is Object Off The Screen?
{
SetObject(loop); // If So, Reassign New Values
}
```

To make the wings flap, we check to see if the flap value is greater than or less than 1.0f and -1.0f. If the wing is greater than or less than those values, we change the flap direction by making fi=-fi.

So if the wings were going up, and they hit 1.0f, fi will become a negative value which will make the wings go down.

Sleep(15) has been added to slow the program down by 15 milliseconds. It ran insanely fast on a friends machine, and I was too lazy to modify the code to take advantage of the timer :)

```
if ((obj[loop].flap>1.0f) || (obj[loop].flap<-1.0f)) // Time To Change Flap Direction?
{
obj[loop].fi=-obj[loop].fi; // Change Direction By Making fi = -fi
}
}

Sleep(15); // Create A Short Delay (15 Milliseconds)

glFlush (); // Flush The GL Rendering Pipeline
}
```

I hope you enjoyed the tutorial. Hopefully it makes loading textures from a resource a lot easier to understand, and texturing triangles a snap. I've reread this tutorial about 5 times now, and it seems easy enough, but if you're still having problems, let me know. As always, I want the tutorials to be the best that they can be, so feedback is greatly appreciated!

Thanks to everyone for the great support! This site would be nothing without it's visitors!!!

Jeff Molofee (**NeHe**)

Appendix 1: Setting Up OpenGL In 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 (www.metrowerks.com). 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 (developer.apple.com/opengl/) 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**)

Appendix 2: Setting Up OpenGL In Solaris

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

The Development Tools:

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

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

The C Compiler:

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

http://www.sunfreeware.com/

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

```
> pkgadd -d gcc-xxxversion
```

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

```
> admintool
```

Browse->Software
Edit->Add

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

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

The OpenGL library

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

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

```
> ls libGL*
```

This should print:

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

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

But are the header files also there?

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

```
> ls
```

This should print:

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

I have it. But what version is it?

This is a FAQ.

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

Helps you with questions dealing with OpenGL on Sun platforms.

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

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

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

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

GLUT

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

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

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/sparc_solar
is/glut-3.7/lib/glut
```

```
> 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/sparc_solaris/glut-
3.7/lib/glut
```

Verify that everything is correct:

```
> echo $LD_LIBRARY_PATH
```

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

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.

Appendix 3: Setting Up OpenGL In MacOS X Using GLUT

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

What You'll Need:

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

Getting Started with Project Builder:

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

Now goto the "Project" Menu and "Add Framework..." to add the GLUT.framework. In 10.1 you need to add the OpenGL.framework as well, so do this now.

Getting Started with GLUT:

Remove the default code by one of two ways:

Delete the main.m file that comes with the default project, and insert a new main.c with the GLUT code or... Select all the code in main.m and replace it with the GLUT code.

You need 3 headers to start with:

```
#include <OpenGL/gl.h> // Header File For The OpenGL32 Library
#include <OpenGL/glu.h> // Header File For The GLu32 Library
#include <GLUT/glut.h> // Header File For The GLut Library
```

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.

All Done!

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

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

Have fun!

R.Goff (unreality@mac.com)

References

Reference Standards

- Mason Woo, OpenGL Architecture Review Board, Jackie Neider, Tom Davis, Dave Shreiner [1999], ***The OpenGL Programming Guide 3rd Edition The Official Guide to Learning OpenGL Version 1.2***

- Opengl Architecture Review Board, Chris Frazier [1997], ***OpenGL Reference Manual 3rd edition***