Introduction

OpenGL has undergone some major changes recently from version 3 and onwards. OpenGL 3 introduced a new way of programming in OpenGL that slightly raises the learning curve, but also does away with the entire Immediate Mode by deprecating a lot of the functions. Now I know I have called this part of the site "OpenGL 4 Tutorials", and all of a sudden I am talking about OpenGL 3, this is because OpenGL 3 and OpenGL 4 are extremely similar, with OpenGL 4 bringing in some new extensions, and because my graphics card only supports OpenGL 3.x. If you would like to see some OpenGL 4 extensions in use, you can always donate and help me get a new graphics card. In this tutorial, I am going to be simply creating a window with an OpenGL 3.2 context attached, nothing more, nothing less. These tutorials are going to be for Windows only, but if you are keen to get them working on a different OS (My Macbook doesn't support OpenGL 3.x), please send me the ports and I will post them up.

OpenGL 3+ Key Differences

An OpenGL 3+ and 4 context requires you to create a standard OpenGL context (version 1.x or 2.x) and then get your application to opt-in to using an OpenGL 4 context.

Without the Immediate Mode, OpenGL has gone for an almost entirely shader driven architecture, which gives us control over everything from the word go, and also greatly improves performance.

When it comes to rendering shapes, instead of setting the state and then calling the vertices one by one, OpenGL 3 and upwards wants us to use Vertex Buffer Objects and Vertex Buffer Arrays straight out. Vertex Buffer Objects were available in OpenGL 1.5 and up as an extension, but they are now a part of the core of OpenGL.

Matrices are one of the major things affected by OpenGL 3. To put it simply, OpenGL does no matrix operations for us such as setting the Projection Matrix, setting the Model Matrix and setting the View Matrix. This means calls such as glRotate and glTranslate are not deprecated and not recommended. OpenGL now likes us to handle our own matrices. I will talk about this in a later tutorial, but there is a library called GLM, designed for use with OpenGL and GLSL, which will help us with our matrix operations.

Coding

Let's finally get into this tutorial, because it is going to be a long one. I'm also going to be aiming for a more Object Oriented approach to programming in these newer tutorials.

This tutorial is going to be very similar in some ways to most Win32 OpenGL tutorials available, the only difference being that it has an OpenGL 3.2 context instead of a standard context. So open up Visual Studio and create a new Windows Console Application in C++. Make sure it is empty to begin with, and then create 4 files. I am calling them: "main.cpp, main.h, opengl_3.cpp, opengl_3.h". I am going to go through these files one by one, starting with the header files, so that we don't get mixed up shuffling between them.

Main.h

Main.h is going to contain all of our header files for the time being. So open it up and paste the following code. I won't explain it, as it is pretty straight forward; we are including some header files

and linking some libraries, the only tricky part may be “#pragma once”, which isn’t standard but is widely supported, and means to only include the file once during compilation.

```
#pragma once

// Include Windows functions
#include <Windows.h>

// Include input and output operations
#include <iostream>

// OpenGL and GLEW Header Files and Libraries
#include <GL/glew.h>
#include <GL/wglew.h>
#pragma comment(lib, "glew32.lib")
#pragma comment(lib, "opengl32.lib")
```

As you can see from the above code, we are once again using GLEW. GLEW is up to date with the latest OpenGL 3.x and possibly 4.0 extensions. I highly recommend the use of it as it makes coding in OpenGL so much easier without having to worry about declaring extensions by yourself.

Opengl_3.h

The first thing we want to include here is our main.h file. So add the line:

```
#include "main.h"
```

Once we have included our main header file which contains access to the GLEW and OpenGL commands, we are going to want to create a new class. I am going to call this class OpenGLContext and it is going to be responsible for creating a context, handling setting up of our scene, resizing of our scene, rendering of our scene, and finally, cleaning up of our scene. Start off with an empty class like so:

```
/**
	OpenGLContext is a class designed to store all of your OpenGL functions and
keep them
	out of the way of your application logic. Here we have the ability to create an
OpenGL
	context on a given window and then render to that window.
*/
class OpenGLContext {
public:

private:

protected:
};
```

Next up, we are going to come up with some protected variables that will keep track of our window. When working with the Win32 API, each Window you see has its own HWND which is the identifier for that window. We also have two different contexts we need to work with, the first is the HDC, which is the device context, and the second is the HGLRC which is an OpenGL rendering context. If we create a variable to handle each of these, we get:

```
class OpenGLContext {
```

```
...

protected:
	HGLRC hrc; // Rendering context
	HDC hdc; // Device context
	HWND hwnd; // Window identifier
};
```

Once we have the information we need for a window and a context, we need to keep track of the width and the height of our window. If we don't keep track of these, then whenever we resize the window, we will find that parts of it are not rendered to, or we render more than we have to, resulting in some of our image being cut off. We are going to add two private variables to keep track of this for us:

```
class OpenGLContext {
...

private:
	int windowWidth; // Store the width of our window
	int windowHeight; // Store the height of our window

...
};
```

Finally we need to come up with some method stubs for controlling our OpenGLContext object. I am going to work with two different constructors and a destructor for some standard methods. Followed by four other methods for working with our context which are going to be called "create30Context (HWND), setupScene(void), reshapeWindow(int, int), renderScene(void)".

```
class OpenGLContext {
public:
	OpenGLContext(void); // Default constructor
	OpenGLContext(HWND hwnd); // Constructor for creating our context given a hwnd
	~OpenGLContext(void); // Destructor for cleaning up our application
	bool create30Context(HWND hwnd); // Creation of our OpenGL 3.x context
	void setupScene(void); // All scene information can be setup here
	void reshapeWindow(int w, int h); // Method to get our window width and height
on resize
	void renderScene(void); // Render scene (display method from previous OpenGL
tutorials)

...
};
```

This is all we are going to need for our opengl_3.h header file for now. So let's go and start on opengl_3.cpp.

Opengl_3.cpp

To give use access to our class methods, you are going to want to include opengl_3.h of course, so that will be line number one for this file.

```
#include "opengl_3.h"
```

Now let's get to building us some methods for our class. I am going to start off with the two constructors and the destructor. Our first constructor, the default one, at the moment does nothing, so you can just create an empty constructor. But the second one will make a call to our create30Context method.

```
/**
	Default constructor for the OpenGLContext class. At this stage it does nothing
	but you can put anything you want here.
*/
OpenGLContext::OpenGLContext(void) {

}

/**
	Constructor for the OpenGLContext class which will create a context given a
windows HWND.
*/
OpenGLContext::OpenGLContext(HWND hwnd) {
	create30Context(hwnd); // Create a context given a HWND
}
```

When we get to our destructor, it will handle the removal of our rendering context and our device context. For this, we make use of wglMakeCurrent, wglDeleteContext and ReleaseDC.

```
/**
	Destructor for our OpenGLContext class which will clean up our rendering
context
	and release the device context from the current window.
*/
OpenGLContext::~OpenGLContext(void) {
    wglMakeCurrent(hdc, 0); // Remove the rendering context from our device context
    wglDeleteContext(hrc); // Delete our rendering context

    ReleaseDC(hwnd, hdc); // Release the device context from our window
}
```

Let's get onto our create30Context method now. This method will take in a HWND value to identify a window, and will create a valid rendering and device context on top of it. Create an empty method for this, and then we will set the HWND for our object, and get a valid HDC. I've made this method a Boolean value, so we can return a false value if we are unable to successfully create a context of any kind. However by default, we will fall back to and OpenGL 2.1 context if unable to create an OpenGL 3.x context.

```
/**
	create30Context creates an OpenGL context and attaches it to the window
provided by
	the HWND. This method currently creates an OpenGL 3.2 context by default, but
will default
	to an OpenGL 2.1 capable context if the OpenGL 3.2 context cannot be created.
*/
bool OpenGLContext::create30Context(HWND hwnd) {
	this->hwnd = hwnd; // Set the HWND for our window

	hdc = GetDC(hwnd); // Get the device context for our window

	return true; // We have successfully created a context, return true
}
```

The next thing we need to do, is create a variable called a PIXELFORMATDESCRIPTOR which determines how many bits of colour and depth we get, along with giving us double buffering and support for OpenGL. To do this, we first need to create a new variable, I am going to call it pfd and then we need to allocate some empty memory for it. Here, 32 bits of colour, gives us R, G, B and A channels, while 24 bits gives us only R, G and B (each channel takes 8 bits). After we have set aside some memory for this variable, we need to set the size of it, which will always be sizeof (PIXELFORMATDESCRIPTOR). Next, we will set the flags which allow double buffering, opengl support and drawing to a window. Next up, we set the iPixelType to allow RGBA pixels, before we set how many bits we want in our color and depth buffers. Finally, we need to set the iLayerType. We are going to set this to the Main_Plane for our window.

```
bool OpenGLContext::create30Context(HWND hwnd) {
	this->hwnd = hwnd; // Set the HWND for our window

	hdc = GetDC(hwnd); // Get the device context for our window

	PIXELFORMATDESCRIPTOR pfd; // Create a new PIXELFORMATDESCRIPTOR (PFD)
	memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR)); // Clear our  PFD
	pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR); // Set the size of the PFD to the
size of the class
	pfd.dwFlags = PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW; //
Enable double buffering, opengl support and drawing to a window
	pfd.iPixelType = PFD_TYPE_RGBA; // Set our application to use RGBA pixels
	pfd.cColorBits = 32; // Give us 32 bits of color information (the higher, the
more colors)
	pfd.cDepthBits = 32; // Give us 32 bits of depth information (the higher, the
more depth levels)
	pfd.iLayerType = PFD_MAIN_PLANE; // Set the layer of the PFD

	return true; // We have successfully created a context, return true
}
```

Next up, we need to choose a pixel format for our device context, based on our pixel format descriptor. To do this, we make a call to ChoosePixelFormat and pass through our hdc and pfd variables, in return we get an integer value that represents our pixel format, which we can then use to set our pixel format for our device context using SetPixelFormat as seen below.

```
bool OpenGLContext::create30Context(HWND hwnd) {
	this->hwnd = hwnd; // Set the HWND for our window

	hdc = GetDC(hwnd); // Get the device context for our window

	...

	int nPixelFormat = ChoosePixelFormat(hdc, &pfd); // Check if our PFD is valid
and get a pixel format back
	if (nPixelFormat == 0) // If it fails
		return false;

	bool bResult = SetPixelFormat(hdc, nPixelFormat, &pfd); // Try and set the
pixel format based on our PFD
	if (!bResult) // If it fails
		return false;

	return true; // We have successfully created a context, return true
```

```
}
```

Once we have our pixel format all set up. We need to go ahead and create our OpenGL context. To do this, we need to first create an OpenGL 2.1 compatible context, and then call a new OpenGL 3.x extension called wglCreateContextAttribsARB to create an OpenGL 3.x compatible context. So let's create our OpenGL 2.1 context as per a standard Win32 OpenGL application.

```
	HGLRC tempOpenGLContext = wglCreateContext(hdc); // Create an OpenGL 2.1 context for our device context
	wglMakeCurrent(hdc, tempOpenGLContext); // Make the OpenGL 2.1 context current and active
```

And now we will go ahead and enable GLEW so that we can use our new OpenGL 3.x context creation extension.

```
	GLenum error = glewInit(); // Enable GLEW
	if (error != GLEW_OK) // If GLEW fails
		return false;
```

To create our context correctly, we need to set some attributes. These include the major and minor version of the OpenGL version we wish to use, followed by a flag to set this as a forward compatible context. When I say major and minor version, OpenGL versions come in this naming convention "major.minor", so in OpenGL 3.2, the major version is 3 and the minor version is 2. This can be change to give OpenGL 3.2, 3.1, 3.0 or even 4.0 contexts.

```
	int attributes[] = {
		WGL_CONTEXT_MAJOR_VERSION_ARB, 3, // Set the MAJOR version of OpenGL to 3
		WGL_CONTEXT_MINOR_VERSION_ARB, 2, // Set the MINOR version of OpenGL to 2
		WGL_CONTEXT_FLAGS_ARB, WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB, // Set our OpenGL context to be forward compatible
		0
	};
```

Now that our attributes are all sorted out, we are going to do a check to see if our OpenGL 3.x context creation extension is available. If it is, we are going to create our OpenGL 3.2 context, if it isn't, then we are going to stick with the OpenGL 2.1 context we created above. To switch to the new context after it has been created, we need to make sure that no context is current on our window, and then delete the old context, before finally making our new OpenGl 3.2 context current.

```
	if (wglewIsSupported("WGL_ARB_create_context") == 1) { // If the OpenGL 3.x context creation extension is available
		hrc = wglCreateContextAttribsARB(hdc, NULL, attributes); // Create and OpenGL 3.x context based on the given attributes
		wglMakeCurrent(NULL, NULL); // Remove the temporary context from being active
		wglDeleteContext(tempOpenGLContext); // Delete the temporary OpenGL 2.1 context
		wglMakeCurrent(hdc, hrc); // Make our OpenGL 3.0 context current
	}
	else {
```

```
            hrc = tempOpenGLContext; // If we didn't have support for OpenGL 3.x and
up, use the OpenGL 2.1 context
        }
```

Finally, to check that we have created our context correctly, we are just going to pull out the MAJOR and MINOR version numbers for OpenGL and output them to a console.

```
        int glVersion[2] = {-1, -1}; // Set some default values for the version
        glGetIntegerv(GL_MAJOR_VERSION, &glVersion[0]); // Get back the OpenGL MAJOR
version we are using
        glGetIntegerv(GL_MINOR_VERSION, &glVersion[1]); // Get back the OpenGL MAJOR
version we are using

        std::cout << "Using OpenGL: " << glVersion[0] << "." << glVersion[1] <<
std::endl; // Output which version of OpenGL we are using
```

On Windows, you won't get a console for a Win32 Application, but a nifty trick to get console output, is to open up Command Prompt, navigate to your directory with your executable file, and use something like: "program.exe > temp.txt"

This will run your application, and save all output to the file temp.txt. Linux programmers may already be familiar with calls like this.

Now we still have three more methods we need to create here before we are done, but luckily for us, they are extremely short. The first I will look at is setupScene, which takes no parameters, and will only set up our OpenGL Clear Color value. I am going to set it to the nice CornFlowerBlue you get when working with XNA. I grew fond of this colour, and it looks nicer than a standard black or white.

```
/**
        setupScene will contain anything we need to setup before we render
*/
void OpenGLContext::setupScene(void) {
        glClearColor(0.4f, 0.6f, 0.9f, 0.0f); // Set the clear color based on
Microsofts CornflowerBlue (default in XNA)
}
```

Moving on, we now have our reshapeWindow method. This method will take in two integers, one for the width of our window and the other for the height of our window. It will then set our OpenGLContext objects width and height to these values.

```
/**
        reshapeWindow is called every time our window is resized, and it sets our
windowWidth and windowHeight
        so that we can set our viewport size.
*/
void OpenGLContext::reshapeWindow(int w, int h) {
        windowWidth = w; // Set the window width
        windowHeight = h; // Set the window height
}
```

And the last method for this file, is our renderScene method, which once again takes no arguments. The first thing we need to do, is set the size of our viewport to the size of our window, which is done

with a call to glViewport. Next we will clear our color, depth and stencil buffers, before finally swapping our front and back buffers to draw to the screen.

```
/**
	renderScene will contain all our rendering code.

	The first thing we are going to do is set our viewport size to fill the entire
window.

	Next up we are going to clear our COLOR, DEPTH and STENCIL buffers to avoid
overlapping
	of renders.

	Any of your other rendering code will go here.

	Finally we are going to swap buffers.
*/
void OpenGLContext::renderScene(void) {
    glViewport(0, 0, windowWidth, windowHeight); // Set the viewport size to fill the
window
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT); //
Clear required buffers
	SwapBuffers(hdc); // Swap buffers so we can see our rendering
}
```

Now we have everything in place for our basic OpenGL 3.2 context, so let's go and finish off this tutorial with the main.cpp file, which will handle all of our Window creation.

Main.cpp

For the purpose of these tutorials, I am going to assume that we are working with single window applications and am not going to go for a more OO approach to window creation. So let's get on to creating our window.

The first thing we need to do here is to include our common header files, main.h and opengl_3.h. Followed by declaring some variables. We are going to need an instance of an OpenGLContext object, a boolean value to say as to whether or not we are still running our application and a standard HINSTANCE variable for Win32. We are also going to give a method declaration for WndProc which is also required for a Windows application.

```
#include "main.h"
#include "opengl_3.h"

OpenGLContext openglContext; // Our OpenGL Context class

bool running = true; // Whether or not the application is currently running

HINSTANCE hInstance; // The HINSTANCE of this application
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Standard window callback
```

The rest of this tutorial, is made up of three methods. The first one here, is required in any Windows application and is responsible for our windows callbacks. We are going to look for calls to resize the window, and for calls to destroy the window. When we find a call to resize the window, we are going to call our openglContext.reshapeWindow method. Lets take a look at the definition for this method.

```
/**
	WndProc is a standard method used in Win32 programming for handling Window
messages. Here we
	handle our window resizing and tell our OpenGLContext the new window size.
*/
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {

	return DefWindowProc(hWnd, message, wParam, lParam);
}
```

In order to look for different window calls, we turn to the message parameter in the method definition. This is a UINT variable, which can be tied to any of the WM_* variables. We are going to need WM_SIZE for window resizing, and WM_DESTROY for calls to destroy the window. I am going to use a simple switch statement to check for these like so:

```
	switch (message) {
		case WM_SIZE: // If our window is resizing
		{

		}

		case WM_DESTROY:
		{

		}
	}
```

When it comes to our window resizing, we are going to need to update our OpenGL context so that it knows the new size of the window. To do this, we can pull out the width and height of the window using calls to LOWORD(lParam) and HIWORD(lParam), which makes things fairly simple for us. Our method now becomes:

```
/**
	WndProc is a standard method used in Win32 programming for handling Window
messages. Here we
	handle our window resizing and tell our OpenGLContext the new window size.
*/
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
	switch (message) {
		case WM_SIZE: // If our window is resizing
		{
			openglContext.reshapeWindow(LOWORD(lParam), HIWORD(lParam)); //
Send the new window size to our OpenGLContext
			break;
		}

		case WM_DESTROY:
		{

		}
	}

	return DefWindowProc(hWnd, message, wParam, lParam);
}
```

The only thing left to check for now, is when a window is destroyed. Then we need to send a call to PostQuitMessage so that we can exit our application.

```
/**
```

```
	WndProc is a standard method used in Win32 programming for handling Window
messages. Here we
	handle our window resizing and tell our OpenGLContext the new window size.
*/
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
	switch (message) {
		case WM_SIZE: // If our window is resizing
		{
			openglContext.reshapeWindow(LOWORD(lParam), HIWORD(lParam)); //
Send the new window size to our OpenGLContext
			break;
		}

		case WM_DESTROY:
		{
			PostQuitMessage(0);
			break;
		}
	}

	return DefWindowProc(hWnd, message, wParam, lParam);
}
```

While we now have a method which will handle all of the call backs for our window, we need to actually go ahead and create a window. So I am going to create a new method called createWindow, which will return a boolean value which determines whether or not the window was created successfully. It's parameters are an LPCWSTR for the title of the window, and two integers, one for the width and one for the height of the window.

```
bool createWindow(LPCWSTR title, int width, int height) {

	return true;
}
```

Once we have our method skeleton in place, we need to setup some variables for our window. We will need a WNDCLASS, HWND and a DWORD. The WNDCLASS, which I will call windowClass sets all the basic information for a window, such as the icon, cursor, type of redraw, the instance of the owner application, etc. The HWND variable which I will just call hwnd, will hold the unique identifier for the window which we create, and finally the DWORD variable, which I am going to call dwExStyle and that just sets the type of window and that it has an edge.

I'm not going to go into detail about all the settings for this, as it's not an OpenGL topic, its a Win32 topic, so here's just the code. I have included a quick error check to make sure the windowClass variable can be registered:

```
bool createWindow(LPCWSTR title, int width, int height) {
	WNDCLASS windowClass;
	HWND hWnd;
	DWORD dwExStyle = WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;

	hInstance = GetModuleHandle(NULL);

	windowClass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
	windowClass.lpfnWndProc = (WNDPROC) WndProc;
	windowClass.cbClsExtra = 0;
	windowClass.cbWndExtra = 0;
```

```
	windowClass.hInstance = hInstance;
	windowClass.hIcon = LoadIcon(NULL, IDI_WINLOGO);
	windowClass.hCursor = LoadCursor(NULL, IDC_ARROW);
	windowClass.hbrBackground = NULL;
	windowClass.lpszMenuName = NULL;
	windowClass.lpszClassName = title;

	if (!RegisterClass(&windowClass)) {
		return false;
	}


	return true;
}
```

Next up, we are going to create our window using the above window class, and then we are going to create our OpenGL context on the window before showing the window and making a quick call to update it.

```
bool createWindow(LPCWSTR title, int width, int height) {
	...

	hWnd = CreateWindowEx(dwExStyle, title, title, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, width, height, NULL, NULL, hInstance, NULL);

	openglContext.create30Context(hWnd); // Create our OpenGL context on the given
window we just created

	ShowWindow(hWnd, SW_SHOW);
	UpdateWindow(hWnd);

	return true;
}
```

This should now give us a window, with an OpenGL 3 context, and it should also be visible. The last thing we need to do is create a WinMain method which will be used to launch our application. Lets create an empty WinMain method, and then we will go ahead and fill this in. Note that I have left in the MSG variable called msg, just because we use it for our return value.

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR    lpCmdLine,
                   int       nCmdShow) {
	MSG msg;

	return (int) msg.wParam;
}
```

Before we fill this in, I am going to give a brief rundown of what to expect. The first few lines are going to be used to convert the title of the window from a char array to an LPCWSTR, they are just an annoying necessity because the CreateWindowEx method doesn't support char arrays. The next two lines after this will create our window, and setup our OpenGL context, and we will end with a loop that will continue running for as long as the application does, and will be responsible for rendering

and checking for window messages. Lets take a look at the first few lines, which convert our char array.

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR    lpCmdLine,
                   int       nCmdShow) {
	MSG msg;

	/**
		The following 6 lines of code do conversion between char arrays and
LPCWSTR variables
		which are used in the Windows API.
	*/
	char *orig = "OpenGL 3 Project"; // Our windows title
	size_t origsize = strlen(orig) + 1;
    const size_t newsize = 100;
    size_t convertedChars = 0;
    wchar_t wcstring[newsize];
    mbstowcs_s(&convertedChars, wcstring, origsize, orig, _TRUNCATE);


	return (int) msg.wParam;
}
```

Now let's make the calls to create our window and setup our OpenGL scene.

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR    lpCmdLine,
                   int       nCmdShow) {
...

	createWindow(wcstring, 500, 500); // Create our OpenGL window

	openglContext.setupScene(); // Setup our OpenGL scene


	return (int) msg.wParam;
}
```

And finally we can set up our infinite loop. We will use the Win32 call to PeekMessage to check for calls to our window. If we get a WM_QUIT message we are going to set our running variable to false. Otherwise we are going to translate and dispatch the message, which is done in our callback. And if we don't have any messages to process, we are going to call our openglContext.renderScene method before returning.

```
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR    lpCmdLine,
                   int       nCmdShow) {
...

	/**
		This is our main loop, it continues for as long as running is true
	*/
	while (running)
	{
```

```
			if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) { // If we have a message
to process, process it
				if (msg.message == WM_QUIT) {
					running = false; // Set running to false if we have a
message to quit
				}
				else {
					TranslateMessage(&msg);
					DispatchMessage(&msg);
				}
			}
			else { // If we don't have a message to process
				openglContext.renderScene(); // Render our scene (which also
handles swapping of buffers)
			}
		}

		return (int) msg.wParam;
}
```

And wow, if you made it this far, give yourself a well deserved pat on the back! It took me a few days to write this tutorial, which is an initial setup on top of which all our other tutorials will build, so once you get it down you are ready to move onto the next. I am also providing Visual Studio 2010 projects for these tutorials, and PDF versions for download. For those of you who have been waiting, I'm sorry it has taken so long, but this is now a 13 page, 4,500+ word introduction to setting up an OpenGL compatible context on a window. If you have any questions you can comment below or email me at swiftless@gmail.com