OpenGL Win32 Tutorial

home: http://www.geocities.com/SiliconValley/Code/1219/
e-mail: craterz@hotmail.com

Introduction

So you wanna do some OpenGL in Windows, eh? Well before you do anything purely OpenGL there's a few Win32 related issues to take care of before you glBegin/glEnd. These issues are addressed here in this document. It's purpose is as a short crash course in how to initialize OpenGL in a Windows program, and possibly a bit more.

I got seriously into OpenGL to play with the beta OpenGL ICD from 3Dfx (Glide isn't half as fun as OpenGL). Nothing is more satisfying in OpenGL than having it hardware accelerated. I also understand that SciTech is in the process of making a OpenGL wrapper that translates everything into DirectX for those without a OpenGL ICD. I'm just hoping that someone finally does away with Direct3D and finally starts using OpenGL.

This is not a OpenGL MFC tutorial. As such, I am not doing any MFC related code. If you want to, port it. The code here is stricly Win32. The code is written and compiled for Visual C++ although the code here is pretty C friendly. In fact, you could probably rename the sample program given at the end from a .CPP to a .C file and it would still work.

Without further delay, let's get to it.

This tutorial is still in the process of being written. As such there is alot todo. If you have questions, comments, etc then feel free to drop me an e-mail. In recent time I've figured out some things I'd like to write down in HTML for myself as well as others. Stuff I'm looking forward to: cameras, texture mapping, call lists, lighting, and more.

Header Files and Linking

Just like any other OpenGL developement in C or C++, the header files are usually stored in the directory gl with-in the primary include directory. As a result, when you go to include the OpenGL header files, it winds up looking like below.

```
#include <gl/gl.h>
#include <gl/glu.h>
```

Most of the time you will only need gl.h. glu.h contains some utility functions that you may find useful.

Note: *In the past I had glaux included. Don't bother, its VERY obselete and rather useless. In fact, I never used it once.*

When it comes time to link the program, make sure you have the associated library files linked. If you just use gl.h then you need atleast opengl32.lib. You may also need to link glu32.lib depending on the program.

Device Context

The first step to initializing OpenGL under Win32 is to grab a device context (DC). Since we're drawing to a window, we simply use the Win32 API function GetDC. There are other alternatives, such as BeginPaint, but GetDC will suffice. We only need the

window's handle (hWnd). Simply make a call to GetDC to get a valid DC.

```
HDC hDC = GetDC( hWnd );
```

Once we're done with OpenGL and any other drawing, we call the function ReleaseDC to tell Windows we're done drawing for now.

```
ReleaseDC( hWnd, hDC );
```

Note: *It was pointed out to me that under some OpenGL ICDs (ATI Rage Pro, RivaTNT, etc.), the window style of CS_OWNDC must be included in the window class. Otherwise OpenGL will not function. This information was provided by Ryan Haksi (thanx Ryan!). I have corrected this in the GLSAMPLE program given at the end.*

Pixel Format

Now that we have a valid DC, there's one more thing todo before we jump into OpenGL. We need to set the pixel format for the DC. This informs the system how we are going to use the DC. Here we specify low-level aspects like double buffering, z-buffer, color format, alpha buffer, etc. We use this my filling the PIXELFORMATDESCRIPTOR structure with information detailing the features we will be needing to use OpenGL.

```
PIXELFORMATDESCRIPTOR pfd;
ZeroMemory( &pfd, sizeof( pfd ) );
pfd.nSize = sizeof( pfd );
pfd.nVersion = 1;
pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
              PFD_DOUBLEBUFFER;
pfd.iPixelType = PFD_TYPE_RGBA;
pfd.cColorBits = 24;
pfd.cDepthBits = 16;
pfd.iLayerType = PFD_MAIN_PLANE;
int iFormat = ChoosePixelFormat( hDC, &pfd );
SetPixelFormat( hDC, iFormat, &pfd );
```

In this case, we've requested OpenGL support and a double buffer. This will suffice for must OpenGL applications. Other options include alpha, accumulation, stencil, and auxiliary buffers. See the Platform SDK for more information on this.

ChoosePixelFormat simply picks the best match for what we requested. SetPixelFormat takes that choice and uses that. Depending on what you are doing, it may be worth it to investigate further these two functions and others related to them.

Double Buffer

Since we're using a double buffered DC, anything we draw to the DC actually goes to a non-visible space in memory. This happens when you specify the PFD_DOUBLEBUFFER flag in the pixel format above. It isn't displayed in the window until we tell it to. This is very useful for rendering the scene off-screen and displaying the final image in one shot. This can be done in one simple function call to SwapBuffers, simply supply the DC in question.

```
SwapBuffers( hDC );
```

And viola! We have our final image displayed.

Render Context

The good news is we only have a couple more function calls before we can start playing with OpenGL. The next step is to create a render context (RC). Just like anything else in Win32, we are dealing with a handle and so we have the HGLRC. This

is our bridge to the OpenGL system. Once we have a DC and set its corresponding pixel format, we can begin by initializing OpenGL and enabling our RC. The first step is to create our OpenGL context using wglCreateContext.

```
HGLRC hRC;
hRC = wglCreateContext( hDC );
```

After we're done working with the RC, we have to delete it to free system resources.

```
wglDeleteContext( hRC );
```

In some cases, there may be several instances of OpenGL render contexts running at the same time, so how's OpenGL supposed to know where the commands are going in an multi-tasking environment? We do this by making our OpenGL RC current using the wglMakeCurrent function.

```
wglMakeCurrent( hDC, hRC );
```

Now we can call all our OpenGL functions to our heart's content. Once we're done, we need to make our rendering context not current. We call the wglMakeCurrent function again with two NULL values.

```
wglMakeCurrent( NULL, NULL );
```

If you feel a little more generous and actually want to preserve the previous current DC and RC, you can use the functions wglGetCurrentDC and

```
wglGetCurrentContext.
HDC hOldDC = wglGetCurrentDC();
HGLRC hOldRC = wglGetCurrentContext();
wglMakeCurrent( hDC, hRC );

// do our OpenGL stuff in between here...

wglMakeCurrent( hOldDC, hOldRC );
```

The difference between this and the NULL approach has todo with if you have multiple RCs in the calling thread. If you only have one, no worries. If you have more, may want to use the safer method.

Putting it Together I: C Style

Now that we can get the window's DC, set the pixel format, create and enable a RC, we can put this all together in two functions to enable and disable OpenGL rendering to a window.

```
void EnableOpenGL(HWND hWnd, HDC * hDC, HGLRC * hRC)
{
    PIXELFORMATDESCRIPTOR pfd;
    int iFormat;

    // get the device context (DC)
    *hDC = GetDC( hWnd );

    // set the pixel format for the DC
    ZeroMemory( &pfd, sizeof( pfd ) );
    pfd.nSize = sizeof( pfd );
    pfd.nVersion = 1;
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                  PFD_DOUBLEBUFFER;
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = 24;
    pfd.cDepthBits = 16;
    pfd.iLayerType = PFD_MAIN_PLANE;
    iFormat = ChoosePixelFormat( *hDC, &pfd );
    SetPixelFormat( *hDC, iFormat, &pfd );

    // create and enable the render context (RC)
    *hRC = wglCreateContext( *hDC );
```

```
    wglMakeCurrent( *hDC, *hRC );
}

void DisableOpenGL(HWND hWnd, HDC hDC, HGLRC hRC)
{
    wglMakeCurrent( NULL, NULL );
    wglDeleteContext( hRC );
    ReleaseDC( hWnd, hDC );
}
```

Now between EnableOpenGL and DisableOpenGL, we can call all the glXXX functions we want. Remember for double buffers that we need the SwapBuffers function above.

This could be compressed into some form of class structure to store the DC and RC in one place, but I'll leave that for later. Of course, there's some error checking that should be added, but that tends to depend on the specific application.

Putting it Together II: C++ Style

Here's the same code above, but in a handy dandy wrapper that hides most of the work and remembering the contexts. This code is not extensively tested and is only done with a single OpenGL render context running.

Call the init function to create the OpenGL context for the window. Then call your OpenGL functions. When done, call the purge function before you destroy the window. How's that for convenience. Now just imagine adding in window creation, message and input handling, etc. Then you have the foundations for a 3D demo or game.

```
class GLContext
{
public:

    GLContext()
    {
        reset();
    }

    ~GLContext()
    {
        purge();
    }

    void init(HWND hWnd)
    {
        // remember the window handle (HWND)
        mhWnd = hWnd;

        // get the device context (DC)
        mhDC = GetDC( mhWnd );

        // set the pixel format for the DC
        PIXELFORMATDESCRIPTOR pfd;
        ZeroMemory( &pfd, sizeof( pfd ) );
        pfd.nSize = sizeof( pfd );
        pfd.nVersion = 1;
        pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                      PFD_DOUBLEBUFFER;
        pfd.iPixelType = PFD_TYPE_RGBA;
        pfd.cColorBits = 24;
        pfd.cDepthBits = 16;
        pfd.iLayerType = PFD_MAIN_PLANE;
        int format = ChoosePixelFormat( mhDC, &pfd );
        SetPixelFormat( mhDC, format, &pfd );

        // create the render context (RC)
        mhRC = wglCreateContext( mhDC );

        // make it the current render context
        wglMakeCurrent( mhDC, mhRC );
    }

    void purge()
    {
        if ( mhRC )
        {
```

```
            wglMakeCurrent( NULL, NULL );
            wglDeleteContext( mhRC );
        }
        if ( mhWnd && mhDC )
        {
            ReleaseDC( mhWnd, mhDC );
        }
        reset();
    }

private:

    void reset()
    {
        mhWnd = NULL;
        mhDC = NULL;
        mhRC = NULL;
     }

    HWND mhWnd;
    HDC mhDC;
    HGLRC mhRC;

};
```

The Hazards of Creation and Destruction

In my experience, trying to create or destroy stuff like DirectX interfaces inside of a WM_CREATE or WM_DESTROY function tends to be rather destructive to one's health: the program, the operating system (namely Windows 95), and the poor soul debugging it. Do stuff like this after you have created the window and it is working correctly. For OpenGL, it seems like having the creation and destruction functions inside the respective WM_ functions works on Windows 98. Still better safe than sorry.

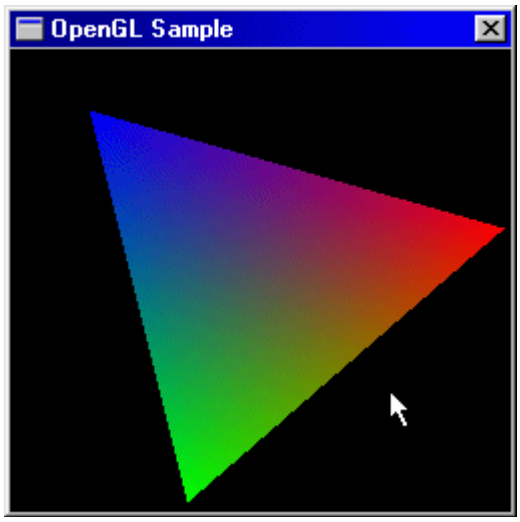
Sample Program

A sample program featuring the two functions given above are available here. You can get the source code and the final program in a handy ZIP file: glsample.zip

A couple things to note so I don't have to answer a ICQ or e-mail that I've already answered 1000 times before:

> Make sure when starting the project/workspace, that you are making a window application (Win32 Application), not a console application. Console applications are nice for GLUT apps, but the example is a Win32 windows aplication, If you get an error saying that linker couldn't find _main, then here is where you messed up.
>
> When linking, make sure you have opengl32.lib linked to the program. The obvious symptom of this is that you get linker errors saying it couldn't find alot of glXXX functions.

Also here's a screen shot of what you can expect to see:



Conclusion

That's about it for now. Later I'd like to cover other Windows aspects like full-screen, experience with 3Dfx's ICD, and such. And maybe even some OpenGL (not just Win32) specifics as well. If you have questions, comments, flames please contact me.

Happy coding!

[powered by Latte and Python | last updated Sat Aug 25 00:54:57 2001 PST]