Using OpenGL in Visual C++

by Alan Oursland

Series Overview

With the release of NT 3.5, OpenGL became a part of the Windows operating system. Now with support for OpenGL in Windows 95 and Windows 98 and low priced graphics accelerators becoming readily available even on low end machines, the prospects of using OpenGL on any Windows machine is becoming more attractive every day. If you are interested in creating quality 2-D or 3-D graphics in Windows, or if you already know another variant of GL, keep reading. This tutorial will show you how to use OpenGL and some of its basic commands.

GL is a programming interface designed by Silicon Graphics. OpenGL is a generic version of the interface made available to a wide variety of outside vendors in the interest of standardization of the language. OpenGL allows you to create high quality 3-D images without dealing with the heavy math usually associated with computer graphics. OpenGL handles graphics primitives, 2-D and 3-D transformations, lighting, shading, Z-buffering, hidden surface removal, and a host of other features. I'll use some of these topics in the sample programs following; others I'll leave to you to explore yourself. If you want to learn more about OpenGL you can search the MSDN website for the keyword "OpenGL".

Here is the list of topics covered in this series:

1. Writing an OpenGL Program
2. Simple 2-D Graphics
3. Transformations and the Matrix Stack
4. Simple 3-D Graphics

Writing an OpenGL Program

The first program demonstrated here will show you the minimum requirements for setting up a Windows program to display OpenGL graphics. As GDI needs a Device Context (DC) to draw images, OpenGL requires a Rendering Context (RC). Unlike GDI, in which each GDI command requires that a DC is passed into it, OpenGL uses the concept of a current RC. Once a rendering context has been made current in a thread, all OpenGL calls in that thread will use the same current rendering context. While multiple rendering contexts may be used to draw in a single window, only one rendering context may be current at any time in a single thread.

The goal of this sample is to create and make current an OpenGL rendering context. There are three steps to creating and making current a rendering context:

1. Set the window's pixel format.
2. Create the rendering context.
3. Make the rendering context current.

Take the following steps to create the project:

1. Create a new Project Workspace of type "MFC AppWizard (exe)". Select the directory you where you want the project directory to be created, and type "GLSample1" as the project name. Click "Create" to enter the AppWizard. Following is a list of the steps in the AppWizard and the parameters you should enter in each of them. Any parameters not listed are optional.
2. Single Document Interface
3. Database support: None
4. Compond Document Support: None

5. Docking Toolbar: OFF (optional)
   Initial Status Bar: OFF (optional)
   Printing an Print Preview: OFF (Printing OpenGL images is accomplished by creating an RC using a printer DC. If you would like to experiment with this later, without rebuilding everything, go ahead and turn this option on).
   Context-Sensitive Help: OFF (optional)
   3D Controls: ON (optional)
6. Use the MFC Standard style of project
   Generate Source File Comments: Yes
   Use the MFC library as a shared DLL.
7. Keep everything at the default.
   Press Finish

Check the "New Project Information" dialog to make sure everything is as it should be and press OK. The new project will be created in the subdirectory "GLSample1".

First we will include all necessary OpenGL files and libraries in this project. Select "Project-Settings" from the menu. Click on the "Link" tab (or press Ctrl-Tab to move there). Select the "General" category (it should already be selected by default), and enter the following into the Object/Library Modules edit box: "opengl32.lib glu32.lib glaux.lib". Press OK. Now open the file "stdafx.h". Insert the following lines into the file:

```
#define VC_EXTRALEAN  // Exclude rarely-used stuff from Windows
                      // headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <gl\gl.h>
#include <gl\glu.h>
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>   // MFC support for Windows 95 Common Controls

#endif // _AFX_NO_AFXCMN_SUPPORT
```

OpenGL requires the window to have styles WS_CLIPCHILDREN and WS_CLIPSIBLINGS set. Edit OnPreCreate so that it looks like this:

```
BOOL CGLSample1View::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= (WS_CLIPCHILDREN | WS_CLIPSIBLINGS);

    return CView::PreCreateWindow(cs);
}
```

The first set to creating a rendering context is to define the window's pixel format. The pixel format describes how the graphics that the window displays are represented in memory. Parameters controlled by the pixel format include color depth, buffering method, and supported drawing interfaces. We will look at some of these below. First create a new protected member function in the CGLSample1View class called "BOOL SetWindowPixelFormat(HDC hDC)" (my preferred method of doing this is right clicking on the class name in the Project Workspace and selecting "Add Function..." from the resulting pop-up menu. You may also do it manually if you wish) and edit the function so that it looks like this:

```
BOOL CGLSample1View::SetWindowPixelFormat(HDC hDC)
{
        PIXELFORMATDESCRIPTOR pixelDesc;
```

```
		pixelDesc.nSize        = sizeof(PIXELFORMATDESCRIPTOR);
		pixelDesc.nVersion    = 1;

		pixelDesc.dwFlags    =    PFD_DRAW_TO_WINDOW |
					      PFD_DRAW_TO_BITMAP |
					      PFD_SUPPORT_OPENGL |
					      PFD_SUPPORT_GDI |
					      PFD_STEREO_DONTCARE;

		pixelDesc.iPixelType        = PFD_TYPE_RGBA;
		pixelDesc.cColorBits        = 32;
		pixelDesc.cRedBits        = 8;
		pixelDesc.cRedShift        = 16;
		pixelDesc.cGreenBits        = 8;
		pixelDesc.cGreenShift        = 8;
		pixelDesc.cBlueBits        = 8;
		pixelDesc.cBlueShift        = 0;
		pixelDesc.cAlphaBits        = 0;
		pixelDesc.cAlphaShift        = 0;
		pixelDesc.cAccumBits        = 64;
		pixelDesc.cAccumRedBits        = 16;
		pixelDesc.cAccumGreenBits    = 16;
		pixelDesc.cAccumBlueBits    = 16;
		pixelDesc.cAccumAlphaBits    = 0;
		pixelDesc.cDepthBits        = 32;
		pixelDesc.cStencilBits        = 8;
		pixelDesc.cAuxBuffers        = 0;
		pixelDesc.iLayerType        = PFD_MAIN_PLANE;
		pixelDesc.bReserved        = 0;
		pixelDesc.dwLayerMask        = 0;
		pixelDesc.dwVisibleMask        = 0;
		pixelDesc.dwDamageMask        = 0;

		m_GLPixelIndex = ChoosePixelFormat( hDC, &pixelDesc);
		if (m_GLPixelIndex==0) // Let's choose a default index.
		{
			m_GLPixelIndex = 1;
			if (DescribePixelFormat(hDC, m_GLPixelIndex,
				sizeof(PIXELFORMATDESCRIPTOR), &pixelDesc)==0)
			{
				return FALSE;
			}
		}

		if (SetPixelFormat( hDC, m_GLPixelIndex, &pixelDesc)==FALSE)
		{
			return FALSE;
		}

		return TRUE;

	}
```

Now add the following member variable to the CGLSample1View class (again, I like to use the right mouse button on the class name and select "Add Variable..."):

```
    int m_GLPixelIndex; // protected
```

Finally, in the ClassWizard, add the function OnCreate in response to a WM_CREATE message and edit it to look like this:

```
int CGLSample1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
                return -1;

        HWND hWnd = GetSafeHwnd();
        HDC hDC = ::GetDC(hWnd);

        if (SetWindowPixelFormat(hDC)==FALSE)
                return 0;

        return 0;
}
```

Compile the program and fix any syntax errors. You may run the program if you wish but at the moment, it will look like a generic MFC shell. Try playing with the pixel format descriptor. You may want to try passing other indices into DescribePixelFormat to see what pixel formats are available. I'll spend some time now explaining what the code does and precautions you should take in the future.

PIXELFORMATDESCRIPTOR contains all of the information defining a pixel format. I'll explain some of the important points here, but for a complete description look in the VC++ online help.

- **`dwFlags`** Defines the devices and interfaces with which the pixel format is compatible. Not all of these flags are implemented in the generic release of OpenGL. Refer to the documentation for more information. dwFlags can accept the following flags:

  PFD_DRAW_TO_WINDOW -- Enables drawing to a window or device surface.
  PFD_DRAW_TO_BITMAP -- Enables drawing to a bitmap in memory.
  PFD_SUPPORT_GDI -- Enables GDI calls. Note: This option is not valid if `PFD_DOUBLEBUFFER` is specified.
  PFD_SUPPORT_OPENGL -- Enables OpenGL calls.
  PFD_GENERIC_FORMAT -- Specifies if this pixel format is supported by the Windows GDI library or by a vendor hardware device driver.
  PFD_NEED_PALETTE -- Tells if the buffer requires a palette. This tutorial assumes color will be done with 24 or 32 bits and will not cover palettes.
  PFD_NEED_SYSTEM_PALETTE -- This flag indicates if the buffer requires the reserved system palette as part of its palette. As stated above, this tutorial will not cover palettes.
  PFD_DOUBLEBUFFER -- Indicates that double-buffering is used. Note that GDI cannot be used with windows that are double buffered.
  PFD_STEREO -- Indicates that left and right buffers are maintained for stereo images.
- **`iPixelType`** Defines the method used to display colors. PFD_TYPE_RGBA means each set of bits represents a Red, Green, and Blue value, while PFD_TYPE_COLORINDEX means that each set of bits is an index into a color lookup table. All of the examples in this program will use PFD_TYPE_RGBA.
- **`cColorBits`** Defines the number of bits used to define a color. For RGBA it is the number of bits used to represent the red, green, and blue components of the color (but not the alpha). For indexed colors, it is the number of colors in the table.
- **`cRedBits, cGreenBits, cBlueBits, cAlphaBits`** The number of bits used to represent the respective components.

- **`cRedShift, cGreenShift, cBlueShift, cAlphaShift`** The number of bits each componet is offset from the beginning of the color.

Once we initialize our structure, we try to find the system pixel format that is closest to the one we want. We do this by calling:

```
m_hGLPixelIndex = ChoosePixelFormat(hDC, &pixelDesc);
```

ChoosePixelFormat takes an hDC and a PIXELFORMATDESCRIPTOR*, and returns an index used to reference that pixel format, or 0 if the function fails. If the function fails, we just set the index to 1 and get the pixel format description using DescribePixelFormat. There are a limited number of pixel formats, and the system defines what their properties are. If you ask for pixel format properties that are not supported, ChoosePixelFormat will return an integer to the format that is closest to the one you requested. Once we have a valid pixel format index and the corresponding description we can call SetPixelFormat. A window's pixel format may be set only once.

Now that the pixel format is set, all we have to do is create the rendering context and make it current. Start by adding a new protected member function to the CGLSample1View class called "BOOL CreateViewGLContext(HDC hDC)" and edit it so that it looks like this:

```
BOOL CGLSample1View::CreateViewGLContext(HDC hDC)
{
        m_hGLContext = wglCreateContext(hDC);
        if (m_hGLContext == NULL)
        {
                return FALSE;
        }

        if (wglMakeCurrent(hDC, m_hGLContext)==FALSE)
        {
                return FALSE;
        }

        return TRUE;
}
```

Add the following member variable to the CGLSample1View class:

```
HGLRC m_hGLContext; // protected
```

Edit OnCreate to call the new function:

```
int CGLSample1View::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
                return -1;

        HWND hWnd = GetSafeHwnd();
        HDC hDC = ::GetDC(hWnd);

        if (SetWindowPixelFormat(hDC)==FALSE)
                return 0;

        if (CreateViewGLContext(hDC)==FALSE)
                return 0;
```

```
        return 0;
}
```

Add the function OnDestroy in response to a WM_DESTROY message and edit it to look like this:

```
void CGLSample1View::OnDestroy()
{
        if(wglGetCurrentContext()!=NULL)
        {
                // make the rendering context not current
                wglMakeCurrent(NULL, NULL) ;
        }

        if (m_hGLContext!=NULL)
        {
                wglDeleteContext(m_hGLContext);
                m_hGLContext = NULL;
        }

        // Now the associated DC can be released.
        CView::OnDestroy();
}
```

And lastly, edit the CGLSample1View class constructor to look like this:

```
CGLSample1View::CGLSample1View()
{
        m_hGLContext = NULL;
        m_GLPixelIndex = 0;
}
```

Once again compile the program and fix any syntax errors. When you run the program it will still look like a generic MFC program, but it is now enabled for OpenGL drawing. You may have noticed that we created one rendering context at the beginning of the program and used it the entire time. This goes against most GDI programs where DCs are created only when drawing is required and freed immediately afterwards. This is a valid option with RCs as well, however creating an RC can be quite processor intensive. Because we are trying to achieve high performance graphics, the code only creates the RC once and uses it the entire time.

CreateViewGLContext creates and makes current a rendering context. wglCreateContext returns a handle to an RC. The pixel format for the device associated with the DC you pass into this function must be set before you call CreateViewGLContext. wglMakeCurrent sets the RC as the current context. The DC passed into this function does not need to be the same DC you used to create the context, but it must have the same device and pixel format. If another rendering context is current when you call wglMakeCurrent, the function simply flushes the old RC and replaces it with the new one. You may call wglMakeCurrent(NULL, NULL) to make no rendering context current.

Because OnDestroy releases the window's RC, we need to delete the rendering context there. But before we delete the RC, we need to make sure it is not current. We use wglGetCurrentContext to see if there is a current rendering context. If there is, we remove it by calling wglMakeCurrent(NULL, NULL). Next we call wglDeleteContext to delete out RC. It is now safe to allow the view class to release the DC. Note that since the RC was current to our thread we could have just called wglDeleteContext without first making it not current. Don't get into the habit of doing this. If you ever start using multi-threaded applications that laziness is going to bite you.

Congratulations on your first OpenGL program, even if it doesn't do much! If you already know OpenGL on another platform then read the tips below and go write the next killer graphics applications. If you don't know OpenGL keep reading. I'll give you a tour of some of its functions.

OpenGL Tips:

1. Set the viewport and matrix modes in response to a WM_SIZE message.
2. Do all of your drawing in response to a WM_PAINT message.
3. Creating a rendering context can take up a lot of CPU time. Only create it once and use it for the life of your program.
4. Try encapsulating your drawing commands in the document class. That way you can use the same document in different views.

Simple 2-D Graphics

The sample program presented in this section will show you how to create the viewport, set up matrix modes, and draw some simple 2-D images.

Start by creating a new project named GLSample2 and setting it up for OpenGL like you did with the first program, or use the first program as your starting point.

Use ClassWizard to add an OnSize function to CGLSample2View in response to a WM_SIZE message. Edit OnSize to look like this:

```
void CGLSample2View::OnSize(UINT nType, int cx, int cy)
{
        CView::OnSize(nType, cx, cy);

        GLsizei width, height;
        GLdouble aspect;

        width = cx;
        height = cy;

        if (cy==0)
                aspect = (GLdouble)width;
        else
                aspect = (GLdouble)width/(GLdouble)height;

        glViewport(0, 0, width, height);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, 500.0*aspect, 0.0, 500.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
}
```

Now use ClassWizard to add an OnPaint function to CGLSample2View in response to a WM_PAINT message. Edit OnPaint to look like this:

```
void CGLSample2View::OnPaint()
{
        CPaintDC dc(this); // device context for painting (added by
                           // ClassWizard)

        glLoadIdentity();
```

```
        glClear(GL_COLOR_BUFFER_BIT);

        glBegin(GL_POLYGON);
                glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
                glVertex2f(100.0f, 50.0f);
                glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
                glVertex2f(450.0f, 400.0f);
                glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
                glVertex2f(450.0f, 50.0f);
        glEnd();

        glFlush();
}
```

Compile and run the program. You should see a black window with a large multicolored triangle in it. Try resizing the window and watch the triangle resize along with it. OnSize defines the viewport and the viewing coordinates. The viewport is the area of the window that the OpenGL commands can draw into. It is set in this program by calling

```
glViewport(0, 0, width, height);
```

This sets the lower left hand corner of the viewport to the lower left hand corner of the window and sets the height and width to that of the window. The parameters passed into the function are in screen coordinates. Try changing the glViewport command in OnSize to the following. Then compile and run the program to see what happens.

```
glViewport(width/4, height/4, width/2, height/2);
```

Make the window taller than it is wide. Because the viewport is smaller than the screen, part of the triangle will be clipped. Change the code back to the way it was originally.

The next command called in OnSize is glMatrixMode(GL_PROJECTION). OpenGL maintains three internal matrices to control various transformations. These matrices are name *Projection*, *ModelView*, and *Texture*. The Projection matrix handles transformations from the eye coordinates to clip coordinates. The ModelView matrix converts object coordinates to eye coordinates. The Texture matrix converts textures from the coordinates they are defined in to the coordinates needed to map them onto a surface. glMatrixMode sets which of these matrices will be affected by matrix operations. Don't worry if you don't understand these right now, I'll explain them as needed.

We call glLoadIdentity to initialize the project matrix. gluOrtho2D sets the project matrix to display a two dimension orthogonal image. The numbers passed into this function define the space within which you may draw. This space is known as the world coordinates. We now initialize the ModelView matrix and leave OpenGL in this matrix mode. Matrix operations (which include transformations) carried out while in the ModelView mode will affect the location and shape of any object drawn. For example if we called "glRotated(30, 0, 0, 1)" just before our glBegin call in OnPaint, our triangle would be rotated 30 degrees around the lower left corner of the screen. We will look at this more a little later. (For those of you who have used IRIS GL, we have just set up the equivalent of calling mmode(MSINGLE). There is an entire section in the VC++ online documentation detailing the differences between IRIS GL and OpenGL for those who are interested.)

OnPaint is the beast that actually draws our triangle. First we clear our ModelView matrix. This isn't really necessary since we aren't doing any transformations, but I added it just in case we decide to do any. Next we clear the color buffer (which in this case happens to be the screen, but could be a print buffer or bitmap depending on the type of device context you used to create rendering context). The next call is glBegin(GL_POLYGON). This function changes the state of the rendering context. From an object oriented perspective, it creates an internal object of type GL_POLYGON, which is defined by all

commands issued until glEnd() is called. We make three glColor4f and three glVertex2f calls to define our triangle.

Let me take a moment at this point to discuss the naming conventions OpenGL uses. All OpenGLcommands use the prefix "gl". There are also a number of "glu" commands which are considered "GL Utilities". These "glu" commands are simply combinations of "gl" commands that perform commonly useful tasks - like setting up 2-D orthographic matrices. Most "gl" commands have a number of variants that each take different data types. The glVertex2f command, for instance, defines a vertex using two floats. There are other variants ranging from four doubles to an array of two shorts. Read the list of glVertex calls in the online documentation and you will feel like you are counting off an eternal list. glVertex2d, glVertex2f, glVertex3i, glVertex3s, glVertex2sv, glVertex3dv...

The definition for our triangle uses the following technique. We call glColor4f(1.0f, 0.0f, 0.0f, 1.0f). This sets the current color to Red by specifying the Red component to 1 and the Green and Blue components to 0. We then define a vertex at point (100,50) in our world coordinates by calling glVertex2f(100.0f, 50.0f). We now have a red vertex at point (100,50). We repeat this process, setting the color to Green and Blue respectively, for the next two vertices. The call to glEnd ends the definition of this polygon. At this point there should still be nothing on the screen. OpenGL will save the list of commands in a buffer until you call glFlush. glFlush causes these commands to be executes. OpenGL automatically interpolates the colors between each of the points to give you the multihued triangle you see on the screen.

Play with some of the different shapes you can create with glBegin. There is a list of modes and valid commands to create shapes below. In the next version of this program, we will move our drawing routines into the document class. I will also show you how to use the basic transforms and the importance of pushing and popping matrices onto and off of the matrix stack.
glBegin(GLenum mode) parameters:

```
GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES,
GL_QUADS, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP,
GL_POLYGON
```

Functions that are valid between glBegin and glEnd:

```
glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord,
glEvalPoint, glMaterial, and glEdgeFlag
```

Transformations and the Matrix Stack

The sample program presented in this section will show you how to use display lists, basic transforms, the matrix stack, and double buffering.

Once again, follow the above steps to get to a starting point for this third sample program (or continue to modify the same program). In this program we will be creating a "robot arm" that you can control with your mouse. This "arm" will actually be two rectangles where one rectangle rotates about a point on the other rectangle. Begin by adding the public member function "void RenderScene(void)" to the CGLSample3Doc class. Modify CGLSample3View::OnPaint and CGLSample3Doc:: RenderScene so that they look like this:

```
void CGLSample3View::OnPaint()
{
        CPaintDC dc(this); // device context for painting

        CGLSample3Doc* pDoc = GetDocument();
```

```
        pDoc->RenderScene();
}

void CGLSample3Doc::RenderScene(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

        glFlush();
}
```

At this time our program generates a black screen. We will do something about that in a minute, but first we need to add some state variables to the CGLSample3Doc class. Add the following enumerated types and variables to the document class. Then initialize them in the document constructor.

```
enum GLDisplayListNames
{
        ArmPart=1
};

double m_transY;
double m_transX;
double m_angle2;
double m_angle1;

CGLSample3Doc::CGLSample3Doc()
{
        m_transY=100;
        m_transX=100;
        m_angle2=15;
        m_angle1=15;
}
```

- ArmPart - This is a identifier for the display list that we will be creating to draw the parts of the arm.
- m_transY - This is the y offset of the arm from the world coordinate system origin
- m_transX - This is the x offset of the arm from the world coordinate system origin
- m_angle2 - This is the angle of the second part of the arm with respect to the first part.
- m_angle1 - This is the angle of the first part of the arm with respect to the world coordinate axis.

We will be using what is known as a display list to draw the parts of our arm. A display list is simply a list of OpenGL commands that have been stored and named for future processing. Display lists are often preprocessed, giving them a speed advantage over the same commands called out of a display list. Once a display list is created, its commands may be executed by calling glCallList with the integer name of the list. Edit CGLSample3Doc::OnNewDocument to look like this:

```
BOOL CGLSample3Doc::OnNewDocument()
{
        if (!CDocument::OnNewDocument())
                return FALSE;

        glNewList(ArmPart);
                glBegin(GL_POLYGON);
                        glVertex2f(-10.0f,  10.0f);
                        glVertex2f(-10.0f, -10.0f);
                        glVertex2f(100.0f, -10.0f);
                        glVertex2f(100.0f,  10.0f);
```

```
                glEnd();
        glEndList();

        return TRUE;
}
```

Note: Microsoft has changed the OpenGL API since this was written. If you are using a newer version of the API, you will need to make the following call to glNewList:

```
glNewList(ArmPart, GL_COMPILE);
```

GL_COMPILE tells OpenGL to just build the display list. Alternatively, you can pass GL_COMPILE_AND_EXECUTE into glNewList. This will cause the commands to be executed as the display list is being built!

Now edit CGLSample3Doc::RenderScene to look like this:

```
void CGLSample3Doc::RenderScene(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

        glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
        glCallList(ArmPart);

        glFlush();
}
```

If you were to run the program now, all you would see is a small red rectangle in the lower left hand corner of the screen. Now add the following lines just before the call to glCallList:

```
glTranslated( m_transX, m_transY, 0);
glRotated( m_angle1, 0, 0, 1);
```

These two commands affect the ModelView matrix, causing our rectangle to rotate the number of degrees stored in m_angle1 and translate by the distance defined by (m_transX, m_transY). Run the program now to see the results. Notice that every time the program gets a WM_PAINT event the rectangle moves a little bit more (you can trigger this by placing another window over the GLSample3 program and then going back to GLSample3). The effect occurs because we keep changing the ModelView matrix each time we call glRotate and glTranslate. Note that resizing the window resets the rectangle to its original position (OnSize clears the matrix to an identity matrix, as you can see in the code) We need to leave the matrix in the same state in which we found it. To do this we will use the matrix stack. Edit CGLSample3Doc::RenderScene to look like the code below. Then compile and run the program again.

```
void CGLSample3Doc::RenderScene(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

        glPushMatrix();
                glTranslated( m_transX, m_transY, 0);
                glRotated( m_angle1, 0, 0, 1);
                glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
                glCallList(ArmPart);
        glPopMatrix();

        glFlush();
}
```

glPushMatrix takes a copy of the current matrix and places it on a stack. When we call glPopMatrix, the last matrix pushed is restored as the current matrix. Our glPushMatrix call preserves the initial identity matrix, and glPopMatrix restores it after we dirtied up the matrix. We can use this technique to position objects with respect to other objects. Once again, edit RenderScene to match the code below.

```
void CGLSample3Doc::RenderScene(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

        glPushMatrix();
                glTranslated( m_transX, m_transY, 0);
                glRotated( m_angle1, 0, 0, 1);
                glPushMatrix();
                        glTranslated( 90, 0, 0);
                        glRotated( m_angle2, 0, 0, 1);
                        glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
                        glCallList(ArmPart);
                glPopMatrix();
                glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
                glCallList(ArmPart);
        glPopMatrix();

        glFlush();
}
```

When you run this you will see a red rectangle overlapping a green rectangle. The translate commands actually move the object's vertex in the world coordinates. When the object is rotated, it still rotates around its own vertex, thus allowing the green rectangle to rotate around the end of the red one. Follow the steps below to add controls so that you can move these rectangles.

1. Add the following member variables to the view class:

```
CPoint m_RightDownPos;     // Initialize to (0,0)
CPoint m_LeftDownPos;    // Initialize to (0,0)
BOOL m_RightButtonDown; // Initialize to FALSE
BOOL m_LeftButtonDown;     // Initialize to FALSE
```

2. Add member functions responding to WM_LBUTTONDOWN, WM_LBUTTONUP, WM_RBUTTONDOWN, and WM_RBUTTONUP. Edit them as shown below:

```
void CGLSample3View::OnLButtonUp(UINT nFlags, CPoint point)
{
        m_LeftButtonDown = FALSE;
        CView::OnLButtonUp(nFlags, point);
}

void CGLSample3View::OnLButtonDown(UINT nFlags, CPoint point)
{
        m_LeftButtonDown = TRUE;
        m_LeftDownPos = point;
        CView::OnLButtonDown(nFlags, point);
}

void CGLSample3View::OnRButtonUp(UINT nFlags, CPoint point)
{
```

```
        m_RightButtonDown = FALSE;
        CView::OnRButtonUp(nFlags, point);
}

void CGLSample3View::OnRButtonDown(UINT nFlags, CPoint point)
{
        m_RightButtonDown = TRUE;
        m_RightDownPos = point;
        CView::OnRButtonDown(nFlags, point);
}
```

3. Add a member function responding to WM_MOUSEMOVE and edit it as shown below.

```
void CGLSample3View::OnMouseMove(UINT nFlags, CPoint point)
{
        if (m_RightButtonDown)
        {
                CGLSample3Doc* pDoc = GetDocument();
                CSize rotate = m_RightDownPos - point;
                m_RightDownPos = point;

                pDoc->m_angle1 += rotate.cx/3;
                pDoc->m_angle2 += rotate.cy/3;
                InvalidateRect(NULL);
        }

        if (m_LeftButtonDown)
        {
                CGLSample3Doc* pDoc = GetDocument();
                CSize translate = m_LeftDownPos - point;
                m_LeftDownPos = point;
                pDoc->m_transX -= translate.cx/3;
                pDoc->m_transY += translate.cy/3;
                InvalidateRect(NULL);
        }

        CView::OnMouseMove(nFlags, point);
}
```

Build and run the program. You may now drag with the left mouse button anywhere on the screen to move the arm, and drag with the right button to rotate the parts of the arm. The above code uses the Windows interface to change data. The OpenGL code then draws a scene based on that data. The only problem with the program now is that annoying flicker from the full screen refreshes. We will add double buffering to the program and then call it complete.

Double buffering is a very simple concept used in most high performance graphics programs. Instead of drawing to one buffer that maps directly to the screen, two buffers are used. One buffer is always displayed (known as the front buffer), while the other buffer is hidden (known as the back buffer). We do all of our drawing to the back buffer and, when we are done, swap it with the front buffer. Because all of the updates happen at once we don't get any flicker.

The only drawback to double buffering is that it is incompatible with GDI. GDI was not designed with double buffering in mind. Because of this, GDI commands will not work in an OpenGL window with double buffering enable. That being said, we first need to change all of the "InvalidateRect(NULL);" calls to "InvalidateRect(NULL, FALSE);". This will solve most of our flicker problem (the rest of the flicker was

mainly to make a point). To enable double buffering for the pixel format, change the pixelDesc.dwFlags definition in CGLSample3View::SetWindowPixelFormat to the following:

```
pixelDesc.dwFlags =    PFD_DRAW_TO_WINDOW |
                        PFD_SUPPORT_OPENGL |
                        PFD_DOUBLEBUFFER |
                        PFD_STEREO_DONTCARE;
```

There are no checks when we set the pixel format to make sure that ours has double buffering. I will leave this as an exercise for the reader.

First we need to tell OpenGL to draw only onto the back buffer. Add the following line to the end of CGLSample3View::OnSize:

```
glDrawBuffer(GL_BACK);
```

Each time we draw a new scene we need to swap the buffer. Add the following line to the end of CGLSample3View::OnPaint:

```
SwapBuffers(dc.m_ps.hdc);
```

When you compile and run the program now you should see absolutely no flicker. However, the program will run noticeably slower. If you still see any flicker then ChoosePixelFormat is not returning a pixel format with double buffering. Remember that ChoosePixelFormat returns an identifier for the pixel format that it believes is closest to the one you want. Try forcing different indices when you call SetPixelFormat until you find a format that supports double buffering.

Simple 3-D Graphics

The sample program presented in this section will show you how to use basic 3-D graphics. It will show you how to set up a perspective view, define and object and transform that object in space. This section assumes some knowledge of graphics. If you don't know what a word means, you can probably look it up in most graphics books. The Foley and Van Dam book listed on this page will definitely have the definitions.

Create an OpenGL window with double buffering enabled. Set up the view class OnSize and OnPaint message handlers just as they are in the previous program. Add a RenderScene function to the document class, but do not put any OpenGL commands into it yet.

First we need to change our viewing coordinate system. gluOrtho2D, the function we have been calling to set up our projection matrix, actually creates a 3 dimensional view with the near clipping plane at z=-1 and the far clipping plane at 1. All of the "2-D" commands we have been calling have actually been 3-D calls where the z coordinate was zero. Surprise! You've been doing 3-D programming all along. To view our cube, we would like to use perspective projection. To set up a perspective projection we need to change OnSize to the following:

```
void CGLSample4View::OnSize(UINT nType, int cx, int cy)
{
        CView::OnSize(nType, cx, cy);

        GLsizei width, height;
        GLdouble aspect;

        width = cx;
```

```
        height = cy;

        if (cy==0)
                aspect = (GLdouble)width;
        else
                aspect = (GLdouble)width/(GLdouble)height;

        glViewport(0, 0, width, height);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45, aspect, 1, 10.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glDrawBuffer(GL_BACK);
}
```

For those who didn't heed my warning above, orthogonal projection maps everything in three-dimensional space onto a two dimensional surface at right angles. The result is everything looks the same size regardless of its distance from the eye point. Perspective project simulates light passing through a point (as if you were using a pinhole camera). The result is a more natural picture where distant objects appear smaller. The gluPerspective call above sets the eye point at the origin, gives us a 45 angle field of view, a front clipping plane at 1, and a back clipping plane at 10.

Now lets draw our cube. Edit RenderScene to look like this:

```
void CGLSample4Doc::RenderScene(void)
{
        glClear(GL_COLOR_BUFFER_BIT);

        glPushMatrix();
                glTranslated(0.0, 0.0, -8.0);
                glRotated(m_xRotate, 1.0, 0.0, 0.0);
                glRotated(m_yRotate, 0.0, 1.0, 0.0);

                glBegin(GL_POLYGON);
                        glNormal3d(  1.0,  0.0,  0.0);
                        glVertex3d(  1.0,  1.0,  1.0);
                        glVertex3d(  1.0, -1.0,  1.0);
                        glVertex3d(  1.0, -1.0, -1.0);
                        glVertex3d(  1.0,  1.0, -1.0);
                glEnd();

                glBegin(GL_POLYGON);
                        glNormal3d( -1.0,  0.0,  0.0);
                        glVertex3d( -1.0, -1.0,  1.0);
                        glVertex3d( -1.0,  1.0,  1.0);
                        glVertex3d( -1.0,  1.0, -1.0);
                        glVertex3d( -1.0, -1.0, -1.0);
                glEnd();

                glBegin(GL_POLYGON);
                        glNormal3d(  0.0,  1.0,  0.0);
                        glVertex3d(  1.0,  1.0,  1.0);
                        glVertex3d( -1.0,  1.0,  1.0);
                        glVertex3d( -1.0,  1.0, -1.0);
                        glVertex3d(  1.0,  1.0, -1.0);
```

```
                glEnd();

                glBegin(GL_POLYGON);
                        glNormal3d(  0.0, -1.0,  0.0);
                        glVertex3d( -1.0, -1.0,  1.0);
                        glVertex3d(  1.0, -1.0,  1.0);
                        glVertex3d(  1.0, -1.0, -1.0);
                        glVertex3d( -1.0, -1.0, -1.0);
                glEnd();

                glBegin(GL_POLYGON);
                        glNormal3d(  0.0,  0.0,  1.0);
                        glVertex3d(  1.0,  1.0,  1.0);
                        glVertex3d( -1.0,  1.0,  1.0);
                        glVertex3d( -1.0, -1.0,  1.0);
                        glVertex3d(  1.0, -1.0,  1.0);
                glEnd();

                glBegin(GL_POLYGON);
                        glNormal3d(  0.0,  0.0, -1.0);
                        glVertex3d( -1.0,  1.0, -1.0);
                        glVertex3d(  1.0,  1.0, -1.0);
                        glVertex3d(  1.0, -1.0, -1.0);
                        glVertex3d( -1.0, -1.0, -1.0);
                glEnd();
        glPopMatrix();
}
```

Add member variables to the document class for m_xRotate and m_yRotate (look at the function definitions to determine the correct type). Add member variables and event handlers to the view class to modify the document variables when you drag with the left mouse button just like we did in the last example (hint: Handle the WM_LBUTTONDOWN, WM_LBUTTONUP, and WM_MOUSEMOVE events. Look at the sample source code if you need help). Compile and run the program. You should see a white cube that you can rotate. You will not be able to see any discernible feature yet since the cube has no surface definition and there is no light source. We will add these features next.

Add the following lines to the beginning of RenderScene:

```
GLfloat RedSurface[]   = { 1.0f, 0.0f, 0.0f, 1.0f};
GLfloat GreenSurface[] = { 0.0f, 1.0f, 0.0f, 1.0f};
GLfloat BlueSurface[]  = { 0.0f, 0.0f, 1.0f, 1.0f};
```

These define surface property values. Once again, the numbers represent the red, green, blue and alpha components of the surfaces. The surface properties are set with the command glMaterial. Add glMaterialCalls to the following locations:

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, RedSurface);
glBegin(GL_POLYGON);
        ...
glEnd();

glBegin(GL_POLYGON);
        ...
glEnd();
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, GreenSurface);
glBegin(GL_POLYGON);
        ...
glEnd();

glBegin(GL_POLYGON);
        ...
glEnd();

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, BlueSurface);
glBegin(GL_POLYGON);
        ...
glEnd();

glBegin(GL_POLYGON);
        ...
glEnd();
```

These new calls make two of the cube faces red, two faces green, and two faces blue. The commands set the ambient color for front and back of each face. However, the cube will still appear featureless until the lighting model is enabled. To do this add the following command to the end of CGLSample4View::OnSize:

```
glEnable(GL_LIGHTING);
```

Compile and run the program. You should see one of the blue faces of the cube. Rotate the cube with your mouse. You will notice the cube looks very strange. Faces seem to appear and disappear at random. This is because we are simply drawing the faces of the cube with no regard as to which is in front. When we draw a face that is in back, it draws over any faces in front of it that have been drawn. The solution to this problem is z-buffering.

The z-buffer holds a value for every pixel on the screen. This value represents how close that pixel is to the eye point. Whenever OpenGL attempts to draw to a pixel, it checks the z-buffer to see if the new color is closer to the eye point than the old color. If it is the pixel is set to the new color. If not, then the pixel retains the old color. As you can guess, z-buffering can take up a large amount of memory and CPU time. The cDepthBits parameter in the PIXELFORMATDESCRIPTOR we used in SetWindowPixelFormat defines the number of bits in each z-buffer value. Enable z-buffering by adding the following command at the end of OnSize:

```
glEnable(GL_DEPTH_TEST);
```

We also need to clear the z-buffer when we begin a new drawing. Change the glClear command in RenderScene to the following:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Compile and run the program to see the results.

We now have a colorful cube that rotates in space and draws correctly, but it is very faint. Let's add a light to the scene so that we can see the cube better. Add the following declaration to the beginning of RenderScene:

```
GLfloat LightAmbient[]    = { 0.1f, 0.1f, 0.1f, 0.1f };
GLfloat LightDiffuse[]    = { 0.7f, 0.7f, 0.7f, 0.7f };
GLfloat LightSpecular[]    = { 0.0f, 0.0f, 0.0f, 0.1f };
GLfloat LightPosition[]    = { 5.0f, 5.0f, 5.0f, 0.0f };
```

These will serve as the property values for our light. Now add the following commands just after glClear in RenderScene:

```
glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmbient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDiffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, LightSpecular);
glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
glEnable(GL_LIGHT0);
```

glLight defines properties for light sources. OpenGL's light sources are all created within the implementation of OpenGL. Each light source has an identifier GL_LIGHT*i* where *i* is zero to GL_MAX_LIGHTS. The above commands set the ambient, diffuse, and specular properties, as well as the position, of light zero. glEnable(GL_LIGHT0) turns on the light.

The program is currently wasting time by drawing the interior faces of the cube with our colored surfaces. To fix this, change the GL_FRONT_AND_BACK parameter in all of the glMaterialfv calls to GL_FRONT. We also want to set the diffuse reflectivity of the cube faces now that we have a light source. To do this, change the GL_AMBIENT parameter in the glMaterialfv calls to GL_AMBIENT_AND_DIFFUSE. Compile and run the program.

You now have a program that displays a lighted, multi-colored cube in three dimensions that uses z-buffering and double buffering. Go ahead and pat yourself on the back. You deserve it.

Conclusion

This concludes the construction of GLSample4 and this tutorial. You should now know how to set up an OpenGL program in Windows, and should also understand some of the basic graphics commands. If you wish to explore OpenGL further, I recommend studying the sample programs in the Microsoft Platform SDK. If you would like to learn more about graphics in general, I recommend the following books. It really is necessary to understand the basics of the material in either of these books if you want to do any serious 3-D graphics.

1. Foley, J. D. and Dam, A. V. and Feiner, S. K. and Hughes., J. F. Computer Graphics, Principles and Practice. Addison-Wesley Publishing Company: Reading, Massachusetts, 1990
2. Hill, F. S. Computer Graphics. MacMillian Publishing Company: New York, 1990.

You may also visit these sites to learm more about OpenGL programming:

- www.sgi.com/software/opengl/
- msdn.microsoft.com/library/default.asp?URL=/library/psdk/opengl/int01_2v58.htm
- Microsoft also offers the following OpenGL articles in msdn.microsoft.com :
  Windows NT OpenGL: Getting Started
  OpenGL I: Quick Start
  OpenGL II: Windows Palettes in RGBA Mode
  OpenGL III: Building an OpenGL C++ Class
  OpenGL IV: Color Index Mode
  OpenGL V: Translating Windows DIBs
  Usenet Graphics Related FAQs
  SAMPLE: MFCOGL a Generic MFC OpenGL Code Sample (Q127071)

I would appreciate any comments or suggestions for this tutorial. Please email me at *naoursla@bellsouth.net*. Thanks!