OpenGL I: Quick Start

Dale Rogerson

Microsoft Developer Network Technology Group

December 1, 1994

Click to open or copy the files in the GLEasy sample application for this technical article.

Abstract

This article describes GLEasy, a simple OpenGL™ program. OpenGL is a three-dimensional (3-D) graphics library included with the Microsoft® Windows NT™ version 3.5 operating system. GLEasy is a Microsoft Foundation Class Library (MFC) application that provides a good starting point for investigations into the Windows NT implementation of OpenGL.

Introduction

There are several methods for teaching something. Take swimming, for example. My Uncle Ulysses had a favorite method for teaching his small young nephews how to swim. This highly sophisticated method of aquatic instruction consisted of tossing one of the aforementioned nephews off the wooden dock and into the ol' fish pond. Uncle U was a big, burly man and could toss a nephew pretty darn far. Unfortunately, I grew up in a different state and had to learn how to swim through a less scientific method of instruction taught by the local YMCA.

Well, Uncle U recently passed on, and in honor of his success in reducing the number of relatives at family reunions, I decided to teach someone something using Uncle U's proven method. The someone is you, and the something is OpenGL™. I'm going to throw you into an OpenGL program. I'll scream some pointers at you from the dock as we go through this program.

In other words, this article focuses on *what* you have to do, not *why* you have to do it. The bibliography provides a list of references that offer information on the "why." This article will teach you how to set up an OpenGL program, so you can devote more time to the fun of rendering three-dimensional (3-D) images. My focus is on the Windows NT™ specifics of getting an OpenGL program set up, not on the actual OpenGL code. Uncle U was similar in this respect. His interest lay in tossing nephews off the dock, not in whether they could swim.

> **Note** Uncle Ulysses is a fictional character. Any resemblance to a real person is coincidental and would definitely scare the willies out of me.

GLEasy

The program I am going to describe is called GLEasy. GLEasy is a simple Microsoft® Foundation Class Library (MFC) application that just happens to include some OpenGL code for placing a cube, a pyramid, and a dodecahedron (a Platonic solid with 12 sides consisting of pentagons) on the screen. You can even rotate these objects. GLEasy is a good place to start playing with OpenGL. Everything is set up for OpenGL to work correctly—all you have to do is add your own OpenGL code to render the scenes you like.

> **Note** In this article, the term "OpenGL" refers to the Windows NT implementation of OpenGL. Some limitations presented in this article are limitations of OpenGL, some are limitations of the generic pixel formats, and other limitations are associated with the Windows NT implementation of OpenGL.

To make it easy for you to follow the discussions in this article, I put all the relevant code in the GLEasy view class **CGLEasyView**, in the GLEASVW.H and GLEASVW.CPP files. I also included the important parts of the code in the article.

The article will follow the flow of the GLEasy program. We will start at the beginning with **PreCreateWindow** and follow the program all the way to **OnDraw**, where our 3-D scene will be rendered. Along the way, we will look at the OpenGL code placed in the following message handlers:

- **PreCreateWindow**
- **OnCreate**
- **OnSize**
- **OnEraseBkgnd**
- **OnInitialUpdate**
- **OnDraw**
- **OnIdle**

First, we need to take care of some logistics.

Logistics

Before you can add OpenGL code to your MFC application, you need to handle the odds and ends described below.

- Add the following lines to STDAFX.H:

  ```
  #include "gl/gl.h"
  #include "gl/glu.h"
  ```

  These lines will add the OpenGL header files for the OpenGL library and the utility library to the precompiled header file. Optionally, you can add the following header file to include the functions defined in the auxiliary library:

  ```
  #include "gl/glaux.h"   // optional
  ```

  The auxiliary library is a collection of routines used by the sample programs listed in the *OpenGL Programming Guide* (called the "Red Book" by the people in the know; see the bibliography at the end of this article for more information). The auxiliary library is great for learning OpenGL, but I wouldn't recommend it for commercial applications. The auxiliary library was written to get something on the screen with as few lines of code as possible. The source code to the auxiliary library is included with the Microsoft Win32® Software Development Kit (SDK) for Windows NT 3.5, but not with Visual C++™.

- Link with the following libraries:

  OPENGL32.LIB

  GLU32.LIB

  GLAUX.LIB (optional)

PreCreateWindow

Now that we have taken care of the logistics, we can start examining the flow of execution in GLEasy. The first member function we call that has some effect on OpenGL is **PreCreateWindow**.

OpenGL can render only into the client area of a window that has been initialized for OpenGL; it cannot render into child windows or siblings of the window. Therefore, we need to make sure that we clip the client area for children and siblings. This is easy enough to do: Simply override **PreCreateWindow** and add the following line:

```
cs.style |= WS_CLIPSIBLINGS | WS_CLIPCHILDREN ;
```

If your window does not have these style bits set, you cannot set a pixel format for it. Pixel formats are discussed in the next section.

OnCreate

Before OpenGL can render images on a drawing surface (window or bitmap), the drawing surface must be initialized. A new concept, *pixel format*, was added to the graphics device interface (GDI) in Windows NT version 3.5. A pixel format specifies several properties of a drawing surface, mainly those dealing with the organization and layout of the color bits. A pixel format is specified with the **PIXELFORMATDESCRIPTOR** structure. Each window has its own current pixel format. Different windows can have different pixel formats, and a single device can support several pixel formats.

For more information on pixel formats, see Dennis Crain's article "Windows NT OpenGL: Getting Started" in the MSDN Library. I would suggest running the MYGL sample application and looking at the pixel formats supported by your system. You can learn a lot about pixel formats by playing around with MYGL for a couple of minutes.

The pixel format must be described, selected, and set for an OpenGL drawing surface before OpenGL commands will work. A good place for setting the pixel format is in the **OnCreate** function for a window. The code in this section is from **CGLEasyView::OnCreate**.

Describing the Pixel Format

First, we must describe the pixel format we would like. To describe a pixel format, we fill in the fields of a **PIXELFORMATDESCRIPTOR** structure. The code below shows how to do just that:

```
CClientDC dc(this) ;
//
// Fill in the pixel format descriptor.
//
PIXELFORMATDESCRIPTOR pfd ;
memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR)) ;
pfd.nSize      = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion   = 1 ;
pfd.dwFlags    = PFD_DOUBLEBUFFER |
                 PFD_SUPPORT_OPENGL |
                 PFD_DRAW_TO_WINDOW ;
pfd.iPixelType = PFD_TYPE_RGBA
pfd.cColorBits = 24 ;
pfd.cDepthBits = 32
pfd.iLayerType = PFD_MAIN_PLANE ;
```

The code above allocates a **PIXELFORMATDESCRIPTOR** structure and fills it in. The **dwFlags** field is set to the following values:

Value	Means
PFD_SUPPORT_OPENGL	We want to use OpenGL on the surface. The pixel format concept is generic enough that interfaces or devices other than OpenGL could use it.
PFD_DRAW_TO_WINDOW	We want to render on the client area of a window and not on a bitmap.
PFD_DOUBLEBUFFER	We want OpenGL to do double buffering for us. OpenGL will render the scene to an off-screen buffer, and we will swap that buffer to the screen.

The **iPixelType** field is set to PFD_TYPE_RGBA. This tells OpenGL that we will specify colors using their red, green, and blue (RGB) components. We could have used PFD_TYPE_COLORINDEX to specify an index into a palette instead of using the RGB components.

The next field of interest is **cColorBits**. This field is set to the number of bits per pixel (bpp), which determines the number of colors. In GLEasy, I set **cColorBits** to 24 because I would like 24 colors if the system supports it. An application can use **GetDeviceCaps** to determine the maximum number of possible colors and choose the bpp value beforehand. Another option is to choose the pixel format you want and see what you actually get. More on choosing the pixel format in a little bit.

While **cColorBits** sets the number of bits we want for color information, **cDepthBits** sets the bpp value for depth information. OpenGL maintains a buffer called the *depth buffer*. For each pixel, the depth buffer contains the distance between the pixel and the viewer. When OpenGL renders an object, it compares the position of each new pixel to the position stored in the depth buffer. If the new pixel is closer to the viewer, it is placed on the screen, and the depth buffer is updated. If the new pixel is farther away from the viewer, it is not written to the screen. As a result, the depth buffer provides a mechanism for hidden surface removal.

Currently, the only **iLayerType** supported is PFD_MAIN_PLANE (the main plane), so I won't go into any more detail on this field.

Choosing and Setting the Pixel Format

After the pixel format descriptor is filled; it is passed to the new Win32 **ChoosePixelFormat** function. This function compares the generic pixel formats supported by Windows NT and any device pixel formats supported by special hardware accelerators with the pixel format you described, and returns the best match. The return value of **ChoosePixelFormat** is a one-based index into the possible pixel formats. The index is not unique and can change depending on the current display mode.

```
int nPixelFormat = ChoosePixelFormat(dc.m_hDC, &pfd);
if (nPixelFormat == 0)
{
   TRACE("ChoosePixelFormat Failed %d\r\n",GetLastError()) ;
   return -1 ;
}
TRACE("Pixel Format %d\r\n",nPixelFormat) ;
```

You are free to examine the pixel format recommended by **ChoosePixelFormat** and choose again if you don't like it. If you like the format recommended by **ChoosePixelFormat**, set the pixel format using the new **SetPixelFormat** function. **SetPixelFormat** takes a handle to a device context (DC) as its first parameter, and sets the window associated with this device context to the appropriate pixel format, as shown below:

```
BOOL bResult = SetPixelFormat (dc.m_hDC, nPixelFormat, &pfd);
if (!bResult)
{
   TRACE("SetPixelFormat Failed %d\r\n",GetLastError()) ;
   return -1 ;
}
```

For more information on describing, selecting, and setting the pixel format, see Dennis Crain's article "Windows NT OpenGL: Getting Started" in the MSDN Library.

Creating a Rendering Context

We have set the pixel format. Next, we need to create an OpenGL *rendering context* (GLRC). You can think of a rendering context as a port through which all OpenGL commands must pass. The rendering context you create has the same pixel format as the device context with which it is associated. A rendering context is not the same as a device context: A device context contains information for GDI, while a rendering context contains information for OpenGL. In many ways, however, a rendering context is to OpenGL what a device context is to GDI. You can create multiple rendering contexts in a program.

To create a rendering context, you use the **wglCreateContext** function. **wglCreateContext** is known as a "wiggle" function. The Windows NT implementation of OpenGL includes several wiggle functions, which are used as bridges to get Windows-specific information, such as the current DCs, into or out of the rendering context.

wglCreateContext returns an HGLRC, which is a handle to the rendering context. In GLEasy, the HGLRC is stored in a member variable, *m_hrc*, of the view class.

```
//
// Create a rendering context.
//
m_hrc = wglCreateContext(dc.m_hDC);
if (!m_hrc)
{
   TRACE("wglCreateContext Failed %x\r\n", GetLastError()) ;
   return -1;
}
```

Quick Look at Palettes

If the PFD_NEED_PALETTE flag in **dwFlags** is set in the pixel format returned from **ChoosePixelFormat**, you need to create a palette. In GLEasy, the **CreateRGBPalette** function creates a palette for the **CGLEasyView** member variable *CPalette m_Pal*. I will not go into much detail about palettes in this article. For more information, read my article "OpenGL II: Windows Palettes in RGBA Mode" in the MSDN Library.

Here are a few facts: If the PFD_NEED_PALETTE flag is set, you have to create a palette. For the generic OpenGL pixel formats, this must be a 3-3-2 palette, which means that the 8 bits are divided into 3 bits for red, 3 bits for green, and 2 bits for blue. The **CGLEasyView::CreateRGBPalette** function creates the palette correctly for OpenGL. Other palettes will result in incorrect colors in pictures rendered by OpenGL. Unless you understand what you are doing (that is, unless you've read my "OpenGL II: Windows Palettes in RGBA Mode" article), you should use **CreateRGBPalette**.

Summary of OnCreate

In **CGLEasyView::OnCreate**, we describe the pixel format we want and give this information to **ChoosePixelFormat**, which finds the closest match to the format we described. We then pass this format to **SetPixelFormat**, which sets the window to the correct format. Now we can create a rendering context with **wglCreateContext** to accept OpenGL commands. The last step is to create a palette, if one is needed.

OnSize

After you create a window, you must size it, so the next area of discussion is **CGLEasyView::OnSize**.

When mapping a 3-D coordinate to the 2-D screen, OpenGL must know the size of the client area. In the **OnSize** function, we set up the transformations needed to map 3-D coordinates to the screen. GDI doesn't know anything about the third dimension, so it is up to OpenGL to set up the 3-D projection.

wglMakeCurrent

A program can have several rendering contexts as well as several device contexts. Before we can draw using GDI, or render using OpenGL, we need to specify the device context or rendering context we are using. GDI and OpenGL have different philosophies on specifying the current context: GDI calls require an explicit device context, while OpenGL calls use an implicit rendering context.

All GDI functions either take a handle to a device context:

```
SetViewport(hDC,x,y)
```

or (with MFC) require a **CDC** object or pointer:

```
dc.SetViewport(x,y) ;
```

OpenGL, on the other hand, adopts the concept of a *current* rendering context. Instead of specifying the rendering context as a parameter for every OpenGL call (as GDI does with device contexts), OpenGL writes to the current active rendering context. The **wglMakeCurrent** function is used to set the current active rendering context:

```
BOOL bResult = wglMakeCurrent (dc.m_hDC, m_hrc);
if (!bResult)
{
   TRACE("wglMakeCurrent Failed %x\r\n", GetLastError() ) ;
   return ;
}
```

If a rendering context is not made current, the OpenGL calls will do nothing. Each thread can have only one current rendering context, and a rendering context can be current in only one thread at a time.

Setting Up the World

Now that we have a current rendering context, we can start using OpenGL functions. The following functions set up the screen.

```
GLdouble gldAspect = (GLdouble) cx/ (GLdouble) cy;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(30.0, gldAspect, 1.0, 10.0);
glViewport(0, 0, cx, cy);
```

All OpenGL functions are prefixed with "gl". The OpenGL utility library includes some convenient medium-level functions built from the low-level functions in the OpenGL library. All OpenGL utility functions (for example, **gluPerspective**) are prefixed with "glu". For portability, OpenGL has its own types, such as **GLdouble**.

I won't describe these functions in detail. Instead, I recommend that you read the Red Book—after all, that's why it was written.

Before drawing with GDI functions, we have to set up our coordinate system. GDI provides functions such as **SetMapMode**, **SetViewportOrg**, **SetViewportExt**, **SetWindowOrg**, and **SetWindowExt** for configuring the coordinate system for your application. For many applications, the default settings of the coordinate system are good enough.

OpenGL applications must also set up their "world," which is complicated by the fact that their world is 3-D. The four functions listed in the code fragment above set up the world. Let's take each one in turn.

glMatrixMode

```
glMatrixMode(GL_PROJECTION);
```

Matrix calculations are heavily used in 3-D graphic programming, and OpenGL is no exception. Matrices are used for transforming (for example, scaling, translating, rotating) objects. Matrices are also used to transform the way information is projected onto the screen. OpenGL maintains two separate transformation matrix stacks: one for transforming objects and the other for transforming the projection of the objects. We want to set up the way we view the objects, so we specify the GL_PROJECTION matrix mode.

glLoadIdentity

```
glLoadIdentity();
```

Transformations are combined mathematically. Any transformation added to the stack is combined with previous transformations. Therefore, the stack must be cleared by loading the identity matrix.

gluPerspective

```
gluPerspective(30.0, gldAspect, 1.0, 10.0);
```

We use the utility library function **gluPerspective** instead of "gl" functions because it is much simpler. **gluPerspective**, as called above, sets the field of view to 30 degrees. The aspect ratio is adjusted for the size of the window client area. The near clipping plane is set at 1 unit from the viewpoint, and the far clipping plane is set at 10 units from the viewpoint. The viewpoint is located at (0, 0, 0) facing down the negative *z* axis, unless we change it (which we don't). *Z*-axis values that are greater than –1 and less than –10 are clipped from the screen.

glViewport

```
glViewport(0, 0, cx, cy);
```

glViewport instructs OpenGL that it is going to render to the whole client area. If you would like to limit rendering to a specific part of the client area, you can set the parameters accordingly.

For more information, check out the three articles by Jeff Prosise in the *Microsoft Systems Journal* (see the bibliography at the end of this article), and read the Red Book, mentioned earlier.

Finishing up OnSize

Before we leave **OnSize**, we call **wglMakeCurrent** to deactivate our current rendering context:

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

This step is not required, but it can help find errors, especially when you are using multiple rendering contexts.

The **CGLEasyView** class could be rewritten to make the window's DC current, and then leave it. More on this later.

OnEraseBkgnd

Overloading **OnEraseBkgnd** and returning TRUE will stop the program from painting the screen white before you render your OpenGL screen. This will make your program look much better because it will eliminate the extra screen flash.

OnInitialUpdate

OpenGL has several tricks for optimizing rendering operations. One trick is to use display lists. A display list is like a metafile; similar to the way a metafile holds GDI commands for later replay, a display list holds OpenGL commands for later replay. When OpenGL builds a display list, it can store the results of its transformation calculations in the list so these calculations do not have to be repeated each time the list is displayed.

A convenient place to build display lists is in the **OnInitialUpdate** member function. The display lists are built before we need them in the **OnDraw** member function.

In GLEasy, the **PrepareScene** helper function contains the code for building the display list for the box and the pyramid.

OnDraw

Now is the time to render the picture. This process consists of the following steps:

Select and realize the palette in the DC.

Make the rendering context current.

Draw the scene using OpenGL commands.

Swap the drawing buffer, if using a double buffering pixel format.

Select the original palette back into the DC.

It is important that you select the palette before you call **wglMakeCurrent**. **wglMakeCurrent** initializes the rendering context based on the current logical palette.

The buffers are swapped using a new Win32 GDI function, **SwapBuffers**.

The code below implements these steps.

```
void CGLEasyView::OnDraw(CDC* pDC)
{
    // Select the palette.
    CPalette* ppalOld = pDC->SelectPalette(&m_Pal, 0);
```

```
    pDC->RealizePalette();

   // Make the HGLRC current.
   BOOL bResult = wglMakeCurrent (pDC->m_hDC, m_hrc);
   if (!bResult)
   {
      TRACE("wglMakeCurrent Failed %x\r\n", GetLastError() ) ;
   }

   // Draw.
   DrawScene() ;

   // Swap buffers.
   SwapBuffers(pDC->m_hDC) ;

   // Select old palette.
   if (ppalOld) pDC->SelectPalette(ppalOld, 0);

   wglMakeCurrent(NULL, NULL) ;
}
```

Because **wglMakeCurrent** can be a time-consuming function, it is often more efficient to call **wglMakeCurrent** only once in a program. However, this requires keeping a DC around for the entire life of the program. In Windows NT, the number of DCs is limited only by memory, and users running Windows NT have plenty of memory. Dennis Crain discusses how to use DCs in detail in "Windows NT OpenGL: Getting Started" in the section "Pulling It All Together." MYGL (Dennis Crain's sample) and GENGL (the sample included in the Win32 SDK for Windows NT 3.5) both set the DC at the beginning of the program and keep it around.

DrawScene

CGLEasy::OnDraw doesn't contain any OpenGL commands; all of the OpenGL commands are in the **CGLEasy::DrawScene** function. In this section, I will explain the functionality of **DrawScene**. Again, for information on basic OpenGL commands, it's best to refer to the Red Book (Chapter 3 has very good information on the effects of transformations).

My goal in writing **DrawScene** was to render an attractive scene while doing the least amount of work possible. Parts of the **DrawScene** code are listed below. Some of the code has been left out to make the point a little clearer; ellipses mark these deletions.

```
void CGLEasyView::DrawScene()
{
   // Set up some colors.
   GLdouble purple[3] = {1.0, 0.14, 0.6667} ;
   .
   .
   .
   // Enable lighting calculations.
   glEnable(GL_LIGHTING) ;
   glEnable(GL_LIGHT0) ;

   // Enable depth calculations.
   glEnable(GL_DEPTH_TEST);

   // Clear the color and depth buffers.
   glClearColor(0.0f, 0.0f, 0.0f, 0.0f) ;
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   // Set the material color to follow the current color
   glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE) ;
   glEnable(GL_COLOR_MATERIAL) ;

   //
   // Change to model view matrix stack.
   //
   glMatrixMode(GL_MODELVIEW);
   .
   .
   .
   //
   // Draw the box.
   //
   .
   .
   .
   //
   // Draw the pyramid.
   //
   glLoadIdentity();
   glTranslated(-0.7, 0.5, -4.5 );
   glRotated(m_angle[Pyramid].cx, 1.0, 0.0, 0.0);
   glRotated(m_angle[Pyramid].cy, 0.0, 1.0, 0.0);
```

```
    glColor3dv(purple) ;
    glCallList(Pyramid) ;

    //
    // Draw the dodecahedron.
    //
    .
    .
    .
    //
    // Flush the drawing pipeline.
    //
    glFlush ();
}
```

DrawScene starts by enabling lighting calculations. For simplicity, I've used the OpenGL defaults. (Chapter 6 in the Red Book contains some good information on lighting effects.) As an experiment, you can disable the lighting calculations and see what happens. OpenGL supports multiple, separate, and independent lights. **glEnable(GL_LIGHT0)** turns on the first light.

glEnable(GL_DEPTH_TEST) enables depth calculations. OpenGL uses the depth buffer for hidden surface removal. For each pixel on the screen, OpenGL keeps track of the distance between the viewport and the object occupying that pixel. If another object wants to write to a particular pixel, the object's distance to the viewport is compared with the distance stored in the depth buffer. The object closest to the viewport is left in the depth buffer and eventually displayed on the screen. **glClear** clears the drawing buffer and the depth buffer.

In OpenGL, surfaces have different material properties. These properties change how light affects an object. For example, some materials reflect light while others absorb light, and some materials consist of one color while reflecting another color. Sometimes, understanding why a blue object is green can be confusing. The situation can get even worse if you have several lights with different colors.

To avoid the trouble of setting up different material properties, I used **glColorMaterial**. Enabling GL_COLOR_MATERIAL causes OpenGL to use the current color for the material properties of a surface, thus simplifying the situation.

3-D graphics rely heavily on matrix mathematics. The 3-D transformations (rotation, scaling, and translation) are expressed mathematically in terms of matrices. OpenGL maintains stacks of matrices that it combines to transform an object. There are two matrix stacks: one for transforming objects in a scene, and the other for transforming the scene onto the screen. As you will remember, we used the projection stack in the **OnSize** function.

Now we want to transform the box and the pyramid for which we created display lists in **OnInitialUpdate**. The first step is to switch to the matrix stack for transforming objects, known as the *model view stack*. **glMatrixMode(GL_MODELVIEW)** changes the current stack to the model view stack.

To transform the pyramid, we clear out the matrix stack by putting the identity matrix on top. If we don't put the identity matrix on the stack, the transformations we add to the stack will be combined with the current transformations on the stack.

Now we can transform the pyramid. Because of the way the math works out, the transformations actually happen in the reverse of the order specified. Therefore, the pyramid is rotated **m_angle[Pyramid].cy** degrees around the *y* axis, then rotated **m_angle[Pyramid].cx** degrees around the *x* axis. Finally, it is translated to the position (–0.7, 0.5, –4.5).

The color of the pyramid is set with the **glColor3dv** command. The three characters at the end of the **glColor*** command name determine the parameter types the command accepts: for example **glColor3dv** takes an array of three doubles. The RGB intensities are specified as doubles between 0.0 and 1.0.

Finally, we can call the display list to render the pyramid for us. The box and dodecahedron are rendered similarly.

We can now flush the OpenGL command pipeline to make sure that all OpenGL commands are processed before we continue. Once again, see the Red Book for more information.

OnDestroy

In the **OnDestroy** member function, we clean up after ourselves by deleting the rendering context we created way back in **OnCreate**:

```
wglMakeCurrent(NULL, NULL) ;
if (m_hrc)
{
   wglDeleteContext(m_hrc) ;
   m_hrc = NULL ;
}
```

OnIdle

You might be interested in how the objects are rotated. When we call **CGLEasyApp::OnIdle**, this function calls **CGLEasyView::Tick**. An array of **CSize** objects, *m_angle*, keeps track of the rotation around the *x* axis and *y* axis for each object. In **Tick**, the **CSize** object for the current array is incremented by 10 degrees.

```
   m_angle[m_RotatingObject].cx += 10 ;
   m_angle[m_RotatingObject].cy += 10 ;
   if (m_angle[m_RotatingObject].cx >= 360)
      m_angle[m_RotatingObject].cx = 0 ;
   if (m_angle[m_RotatingObject].cy >= 360)
      m_angle[m_RotatingObject].cy = 0 ;
```

The window is then invalidated so that the scene can be redrawn. This causes the following lines to execute, thus drawing the pyramid (if it is the currently rotating object) with a different rotation:

```
   glRotated(m_angle[Pyramid].cx, 1.0, 0.0, 0.0);
   glRotated(m_angle[Pyramid].cy, 0.0, 1.0, 0.0);
```

The rotation performance is not very impressive. In fact, it's pretty poor on my 66 Mhz. Pentium™ system unless you make the scene smaller. In a future article, I will explain how I was able to optimize **GLEasy**.

Conclusion

OpenGL is a powerful 3-D graphics library, and GLEasy is a good place to start your investigation of OpenGL. You can extend GLEasy to add more lighting effects, atmospheric effects, and more objects. See the bibliography below for more information on OpenGL.

Bibliography

Crain, Dennis. "Windows NT OpenGL: Getting Started." April 1994. (MSDN Library, Technical Articles)

Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Reading, MA: Addison-Wesley, 1993. ISBN 0-201-63274-8. (This book is also known as the "Red Book".)

OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1*. Reading, MA: Addison-Wesley, 1992. ISBN 0-201-63276-4. (This book is also known as the "Blue Book".)

Prosise, Jeff. "Advanced 3-D Graphics for Windows NT 3.5: Introducing the OpenGL Interface, Part I." *Microsoft Systems Journal* 9 (October 1994). (MSDN Library Archive Edition, Books and Periodicals)

Prosise, Jeff. "Advanced 3-D Graphics for Windows NT 3.5: The OpenGL Interface, Part II." *Microsoft Systems Journal* 9 (November 1994). (MSDN Library Archive Edition, Books and Periodicals)

Prosise, Jeff. "Understanding Modelview Transformations in OpenGL for Windows NT." *Microsoft Systems Journal* 10 (February 1995).

Rogerson, Dale. "OpenGL II: Windows Palettes in RGBA Mode." December 1994. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL III: Building an OpenGL C++ Class." January 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL IV: Color Index Mode." January 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL V: Translating Windows DIBs." February 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL VI: Rendering on DIBs with PFD_DRAW_TO_BITMAP." April 1995. (MSDN Library, Technical Articles)

Rogerson, Dale. "OpenGL VII: Scratching the Surface of Texture Mapping." May 1995. (MSDN Library, Technical Articles)