OpenGL III: Building an OpenGL C++ Class

Dale Rogerson
Microsoft Developer Network Technology Group

January 18, 1995

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

Click to open or copy the files in the GLlib DLL for this technical article.

Abstract

This article discusses the development of a C++ class library for encapsulating OpenGL™ code. The C++ class presented is for demonstration and educational purposes only. I will expand the class library for future OpenGL articles. The class library is not currently part of the Microsoft® Foundation Class Library (MFC), and there are no plans to add this class to MFC in the future. I assume that the reader has already read the first article in this series, "OpenGL I: Quick Start," in the MSDN Library. The class library is in the GLlib.DLL file included with this article. The EasyGL sample application, also included with this article, uses the classes in GLlib.DLL.

Introduction

C++ classes make programming simpler by hiding complexity. For example, a C++ class can set up a correct palette using the **Create** member function. Using a C++ class is much simpler than trying to understand all the discussions in my article "OpenGL II: Windows Palettes in RGBA Mode" in the MSDN Library.

However, if you're trying to understand a topic such as OpenGL™, using a class library may actually hinder your learning process. Trying to decipher what the class is doing can become a chore, precisely because the class hides information. For example, a C++ class would set pixel formats using a **Create** member function, which requires parameters such as PFD_DOUBLE_BUFFER, PFD_TYPE_RGBA, PFD_DRAW_TO_WINDOW, and PFD_SUPPORT_OPENGL. To understand how the pixel format is set, we need to look not only at the code for the **Create** member function, but also at the code that calls **Create**. Thus, if you're trying to understand the code, you must dig around in various source files.

For this reason, I used an "inline" approach with the GLEasy sample application by writing most of the code directly in message handlers such as **OnSize** and **OnCreate**. To follow what is happening in GLEasy, simply start with **CGLEasyView::OnCreate** and follow the source. You will need to understand only a few functions, such as **PrepareScene** and **DrawScene**. Because the code is presented inline in GLEasy, I believe the OpenGL beginner will have an easier time learning OpenGL from GLEasy than from a C++ class library.

However, inline coding takes you only so far. (Otherwise we wouldn't have functions, now would we?) Once you understand **PIXELFORMATDESCRIPTOR**, **ChoosePixelFormat**, **SetPixelFormat**, **wglCreateContext**, **wglMakeCurrent**, and palettes, there is no need to keep that code in sight. Therefore, a C++ class can make the code easier to use and reuse.

I had another motivation for placing the OpenGL code in a C++ class. For my OpenGL article series, I wanted to demonstrate additional OpenGL topics such as color index mode, optimization, and rendering to a bitmap. For this purpose, I created several sample applications, all of which use the same code to initialize OpenGL. I created a shared class so I wouldn't have to fix the same bug in several locations. However, I did have to ensure that the shared code was backward-compatible or make any necessary changes to the earlier samples.

At first glance, it's difficult to tell which is better: fixing the same bug in multiple locations or ensuring backward compatibility. The compiler is more likely to inform you if your changes break old code than remind you to fix a bug in all of your applications. Therefore, it's better, in general, to ensure backward compatibility and fix bugs in one place.

In this article, I will describe **CGL**, which is the C++ class I will use in my future articles on OpenGL. In

this article, I will cover the following topics:

- My design goals for **CGL**
- The architecture of **CGL**
- Issues involved in placing **CGL** in a dynamic-link library (DLL)
- How to use **CGL** in your own application
- Simplifying the use of **CGL** by adding its own view class, **CGLView**
- Using **CGL** and **CGLView** together in your own application
- A discussion of the EasyGL sample application

CGL Design Goals

All projects, whether they pertain to building programs or flying carpets, require goals. The list below contains some of my design goals for the OpenGL class, **CGL**. Note that these goals may differ from the goals for a Microsoft Foundation Class Library (MFC) OpenGL class, from the goals for Open Inventor (which is a standard C++ class library for OpenGL), and from your own goals for an OpenGL class.

- Don't hide OpenGL code.

  This may sound like a contradiction, but **CGL** should not hide or alter OpenGL code. The purpose of my articles and sample applications is to teach programmers how to use OpenGL on a Windows NT™ system. Therefore, I don't want to disguise OpenGL code. I want to be able to cut and paste OpenGL code from applications that don't use **CGL** into applications that use **CGL**. For example, I should be able to add OpenGL code from the *OpenGL Programming Guide* (also called the Red Book; see the bibliography at the end of this article) into my application, and it should work. An example of a class that doesn't follow this guideline is the **CDC** class in MFC. If you have graphics device interface (GDI) drawing code written in C, it must be converted to use the **CDC** class.

- Hide Windows NT OpenGL implementation details.

  CGL should not hide or change standard OpenGL code, but it should hide the Windows NT implementation details that are not standard across platforms. I want **CGL** to encapsulate the resource context, the device context, and the palette required by all OpenGL programs. Thus, I would like **CGL** to be the only class that contains Windows NT OpenGL implementation details. The standard OpenGL code will be placed in a different class.

- Follow the programming model used by the auxiliary library.

  Using a programming model similar to that used by the OpenGL auxiliary library may help developers who use the Red Book make the transition from C to C++ and MFC. This does not mean that **CGL** will require or use the auxiliary library, but only that the auxiliary library has already established a proven way to hide implementation details without hiding OpenGL details.

- Reduce the effort of making new OpenGL applications.

  I will need several sample applications for the articles I plan to write about OpenGL. I want to make my job of building new samples as simple as possible. If I succeed, my design will also make your job of building OpenGL applications easier.

- Write code that can be shared between sample applications.

  I would like my sample applications to share as much code as possible. I want to fix my bugs in a single location.

- Demonstrate how to make an OpenGL class.

  In the process of building **CGL**, I would like to teach you one way to make an OpenGL class. This will help you when you build your own class, even if you decide to do it differently. (In fact, it may help you decide to build your class differently.)

Class Architecture

When designing the OpenGL class, I considered two architectures: function encapsulation and structure

encapsulation. These architectures are not mutually exclusive, because the function encapsulation method can be part of a structure encapsulation architecture.

Function Encapsulation

Function encapsulation is more "MFC-like" than structure encapsulation. In function encapsulation, the OpenGL functions become member functions in the OpenGL class. These functions are called through an OpenGL class object. Take the following OpenGL code:

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

With function encapsulation, this code could become:

```
CGL gl ;
gl.glMatrixMode(GL_PROJECTION) ;
gl.glLoadIdentity() ;
gl.gluPerspective(30.0, gldAspect, 1.0, 10.0) ;
gl.glViewport(0, 0, cx, cy) ;
```

or even:

```
CGL gl ;
gl.MatrixMode(GL_PROJECTION) ;
gl.LoadIdentity() ;
gl.Perspective(30.0, gldAspect, 1.0, 10.0) ;
gl.Viewport(0, 0, cx, cy) ;
```

Function encapsulation has many benefits, including the following:

- It draws a parallel with the **CDC** class.

  Function encapsulation makes the OpenGL commands look more like the **CDC** member functions that MFC already contains.

- It makes OpenGL commands appear more object-oriented.

  C++ users like calling functions via objects. In function encapsulation, OpenGL commands become members of a C++ class. The OpenGL commands are called through objects, thus they appear to be more object-oriented.

- Through function overloading, it reduces the number of command types.

  OpenGL commands take a variety of parameter types. For example, **glColor** has many forms, including **glColor3b**, **glColor3d**, **glColor3i**, **glColor4f**, and **glColor4uiv**. By using C++ function overloading, you can reduce these forms to a single function: **glColor** or **Color**.

- It encapsulates functions that take floats, so they can take doubles.

  Many OpenGL commands take floats. By default, the Microsoft Visual C++™ version 2.0 compiler treats floating-point literals as doubles, and issues a warning if you pass a double for a parameter that takes a float. To avoid the warning, you must append the letter "f" to the number. For example, **glClearColor(0.0, 0.0, 0.0, 0.0)** will generate a warning, but **glClearColor(0.0f, 0.0f, 0.0f, 0.0f)** will not. You could build versions of these functions that take doubles instead of floats and do the casting for you.

- It provides a way to track errors.

  Function encapsulation provides a way to track errors. We could implement debug versions of all OpenGL commands. These debug versions would print out the error messages that they encounter.

However, the function encapsulation method also has some serious drawbacks:

- It hides OpenGL syntax.

  The purpose of **CGL** is to make learning standard OpenGL easier, not to hide OpenGL from the developer. Full encapsulation of OpenGL commands will make using OpenGL easier, but could make learning standard OpenGL more difficult or confusing. I want you to learn standard OpenGL, not my own dialect of OpenGL.

- It results in loss of portability.

  Because function encapsulation requires changes in syntax, you cannot paste OpenGL code from other sources to your application. Moreover, you cannot paste the code from your application to another application that doesn't use the same class.

- It requires lots of time.

  Building the required function prototypes for all functions in OpenGL takes a lot of time—time I could spend researching another topic or doing something equally productive.

Function encapsulation has another, more serious drawback that I will discuss in the next section.

wglMakeCurrent

In the OpenGL implementation for Windows NT, OpenGL commands require a current active rendering context (RC). Without a current active RC, OpenGL commands do nothing. The **wglMakeCurrent** function makes an RC current. (For more information on **wglMakeCurrent**, see the "OpenGL I: Quick Start" and "Windows NT OpenGL: Getting Started" articles in the MSDN Library.) The most efficient way to use **wglMakeCurrent** is to call it once at the beginning of a program. This method requires keeping a device context (DC) around for the life of the program.

Let's assume that an application renders two separate OpenGL scenes: one scene in the status bar, and the other scene in the client area. Each scene would have a separate rendering context. It would be tempting to write the following code:

```
CGL glStatusBar ;
CGL glClientArea ;

glStatusBar.Color(1.0, 0.0, 0.0) ;
glClientArea.Color(0.0, 1.0, 0.0)  ;
glStatusBar.CallList(STATUS_BAR) ;
glClientArea.CallList(CLIENT_AREA) ;
```

However, for this code to work, each function (**Color** and **CallList**) must call **wglMakeCurrent**, which results in a loss of performance. These functions must at least check to see whether the current RC is correct:

```
void CGL::Color(GLdouble r, GLdouble g, GLdouble b)
{
   if (m_hrc != wglGetCurrentContext())
      wglMakeCurrent(m_pdc->m_hDC, m_hrc) ;
   glColor3d(r, g, b) ;
}
```

The performance loss is more evident if we must call **wglMakeCurrent** explicitly instead of having each member function call it implicitly. A **CGL::MakeCurrent** member function could do the work for us:

```
glStatusBar.MakeCurrent() ;             // RC for glStatus.
glStatusBar.Color(1.0, 0.0, 0.0) ;

glClientArea.MakeCurrent() ;            // Change to RC for glClientArea.
glClientArea.Color(0.0, 1.0, 0.0)  ;

glStatusBar.MakeCurrent() ;             // Change to RC for glStatus.
glStatusBar.CallList(STATUS_BAR) ;

glClientArea.MakeCurrent() ;            // Change to RC for glClientArea.
glClientArea.CallList(CLIENT_AREA) ;
```

A performance loss would result even if we reordered the statements as follows:

```
glStatusBar.MakeCurrent() ;             // Change to RC for glStatus.
glStatusBar.Color(1.0, 0.0, 0.0) ;
glStatusBar.CallList(STATUS_BAR) ;

glClientArea.MakeCurrent() ;            // Change to RC for glClientArea.
glClientArea.Color(0.0, 1.0, 0.0)  ;
glClientArea.CallList(CLIENT_AREA) ;
```

We don't want to hide the OpenGL code, but we should hide the Windows NT functions, because they are not portable to other systems. The OpenGL programmer shouldn't worry about setting the current

context—the **CGL** class should be responsible for this.

I described the scenarios above as extreme examples to illustrate the possible performance loss. Although you might not code in this style intentionally, the use of function encapsulation may indirectly lead to the performance loss illustrated in these examples. A better approach would be to design the class correctly from the beginning, using structure encapsulation (see the next section).

A possible solution to the **wglMakeCurrent** problem that does not require structure encapsulation does exist: Only one RC can be active per thread; therefore, each instance of **CGL** could create a thread and make the RC current in this thread. This solution increases the complexity of **CGL** significantly. The overhead of thread switching may also be greater than the overhead you incur using the **if** statement and **wglGetCurrentContext** function.

Structure Encapsulation

In structure encapsulation, the structure of an OpenGL program is encapsulated within the class, hiding the Windows NT implementation details. The individual OpenGL commands are not provided as member functions of the OpenGL class. (Function encapsulation, on the other hand, includes the OpenGL commands as member functions.)

In the samples associated with my OpenGL articles, I decided to implement structure encapsulation because I felt that function encapsulation was far too time-consuming and offered only syntactic, insignificant benefits.

You may remember that the GLEasy sample application included OpenGL code in the **OnCreate**, **OnSize**, **OnInitialUpdate**, **OnDraw**, and **OnDestroy** functions. Because we are encapsulating the structure of an OpenGL program, it would make sense to include member functions that correspond to those functions in our OpenGL class. A simplified class definition of **CGL** would look like this:

```
class CGL
{
   public:
      BOOL Create() ;
      BOOL Init() ;
      BOOL Resize() ;
      BOOL Render() ;
      void Destroy() ;
}
```

The member functions are called by the appropriate message handler in the view class. Figure 1 illustrates the process.

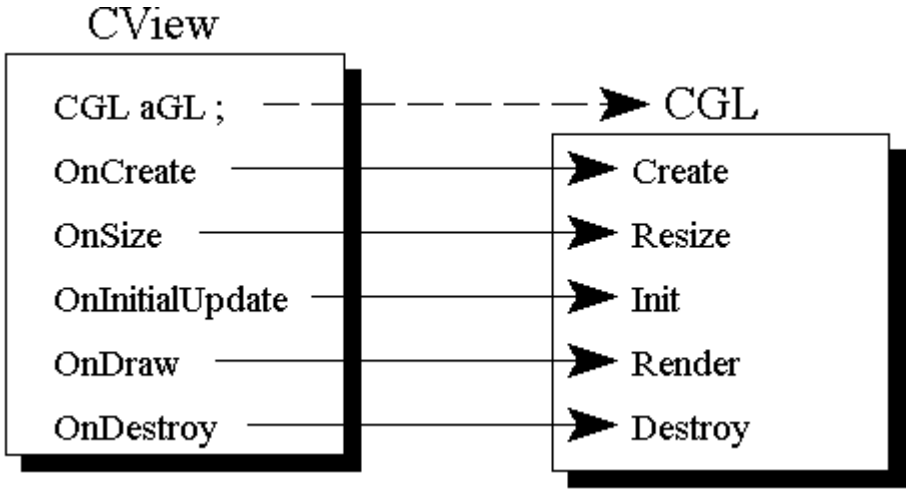

Figure 1. Message handlers in CView call member functions in CGL.

In my design, **CGL::Create** creates a rendering context, a device context, and (if needed) a palette. **CGL::Destroy** handles cleanup tasks for the class. **CGL::Resize** changes the projection of the scene to the screen. **CGL::Init** initializes the OpenGL parameters and sets up display lists. **CGL::Render** puts it all on the screen.

This is very similar to the approach taken by the auxiliary library in the Red Book (see the bibliography at the end of this article). The sample code in the Red Book includes three functions—**myinit**, **myReshape**, and **display**—that parallel **CGL::Init**, **CGL::Resize**, and **CGL::Render**, respectively.

Now, look at **CGLEasyView::OnSize** from the GLEasy sample application:

```
void CGLEasyView::OnSize(UINT nType, int cx, int cy)
{
   CView::OnSize(nType, cx, cy);
   if ( (cx <= 0) || (cy <= 0) ) return ;
   CClientDC dc(this) ;
   BOOL bResult = wglMakeCurrent(dc.m_hDC, m_hrc);
   if (!bResult)
   {
      TRACE("wglMakeCurrent Failed %x\r\n", GetLastError() ) ;
      return ;
   }
   //
   // Set up the 3-D mapping to screen space.
   //
   GLdouble gldAspect = (GLdouble) cx/ (GLdouble) cy;
   glMatrixMode(GL_PROJECTION); OutputGlError("MatrixMode") ;
   glLoadIdentity();
   gluPerspective(30.0, gldAspect, 1.0, 10.0);
   glViewport(0, 0, cx, cy);
   wglMakeCurrent(NULL, NULL);
}
```

The OpenGL code at the end of the function is specific to GLEasy. We wouldn't want this code in **CGL**, so we must separate the application-specific OpenGL code from the common OpenGL code. It just happens that the common OpenGL code is the Windows NT implementation code. In the case above, we can encapsulate the **wglMakeCurrent** function.

The application-specific code is placed in a class inherited from **CGL**, as illustrated in Figure 2.

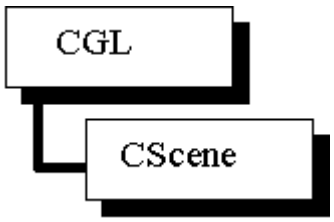

Figure 2. Specific OpenGL code is placed in derived classes.

The most obvious method would be to define **CGL**, the derived class, and **CScene::Resize** as listed below.

CGL:

```
class CGL
{
   public:
      BOOL Create() ;
      virtual BOOL Init() ;
      virtual BOOL Resize(int cx, int cy) ;
      virtual BOOL Render() ;
      void Destroy() ;
}
```

Derived class:

```
class CScene : public CGL
{
   protected:
      virtual BOOL Resize(int cx, int cy) ;
      virtual BOOL Init() ;
      virtual BOOL Render();
}
```

CScene::Resize:

```
BOOL CScene::Resize(int cx, int cy)
{
```

```
   CGL::Resize(cx,cy) ;

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

However, I didn't do it this way. The **Render** function must call **wglMakeCurrent** before, and **SwapBuffer** after, the application-specific OpenGL code. The approach I took mirrors the MFC **CView::OnPaint** function, which calls **BeginPaint** before calling **CView::OnDraw**, and calls **EndPaint** afterwards. So **Render** (a non-virtual function) calls **OnRender** (a virtual function), which contains the application-specific OpenGL code. Instead of making the **Render** function different from the other functions, I decided to give all the functions a similar structure.

Here's a simplified version of the **CGL** class definition:

```
class CGL
{
   public:
      BOOL Create() ;
      BOOL Init() ;
      BOOL Resize() ;
      BOOL Render() ;
      void Destroy() ;
   protected:
      virtual BOOL OnResize() = 0 ;
      virtual BOOL OnInit()   = 0 ;
      virtual BOOL OnRender() = 0 ;
}
```

We derive a class from **CGL** and implement the pure virtual functions:

```
class CScene : public CGL
{
   protected:
      virtual BOOL OnResize() ;
      virtual BOOL OnInit() ;
      virtual BOOL OnRender();
}
```

Figure 3 illustrates the program structure.

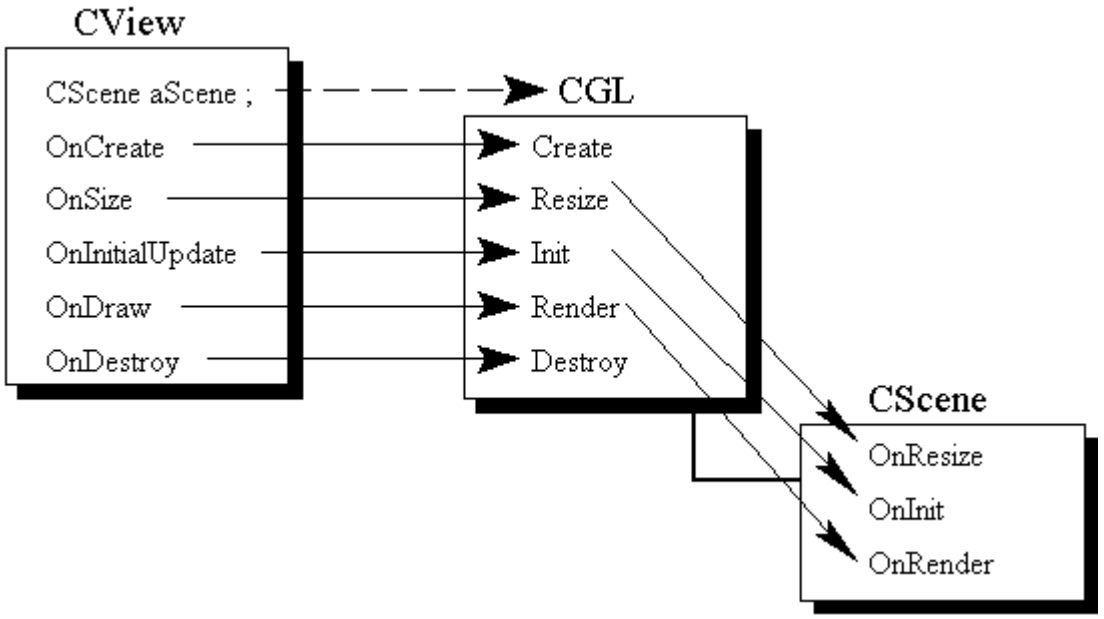

Figure 3. Path of execution from CGL to derived class.

The code for **CGL::Render** looks like this:

```
BOOL CGL::Render()
{
```

```
   // Make the HGLRC current.
   makeCurrent() ;

   // Draw.
   OnRender() ;

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

   return TRUE ;
}
```

wglMakeCurrent

CGL encapsulates the **wglMakeCurrent** function call. **CGL** makes sure that the rendering context of the instance is current before calling any application-specific OpenGL code. **OnResize**, **OnInit**, and **OnRender** all call **CGL::MakeCurrent**, which is a member function implemented in the CGL-HELP.CPP file. The code for **CGL::MakeCurrent** is shown below:

```
void CGL::MakeCurrent()
{
   ASSERT(m_hrc) ;
   ASSERT(m_pdc) ;

   if (m_pPal)
   {
      m_dc->SelectPalette(m_pPal, 0) ;
      m_dc->RealizePalette() ;
   }

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

To save time, **wglGetCurrentContext** checks to see whether the rendering context needs to be changed, and changes it if it does.

CGL in a DLL

Now, I should know better than to ask my colleague Ruediger for his opinions. Ruediger likes doing things the hard way. Don't get me wrong—he doesn't tell you to write a mail merge program as a virtual device driver (VxD), but he does like writing VxDs whenever possible. In fact, he's a happy man when he can write VxDs in hand-assembled machine language while floating on his sea kayak.

When I asked Ruediger's opinion on how to allow multiple applications to share **CGL**, he immediately said, "Use a DLL." Now, I used to love DLLs until C++ came along. C++ classes complicate the DLL interface. MFC extension DLLs (AFXDLLs), on the other hand, simplify exporting C++ classes from a DLL. Although the MFC extension DLLs do not solve all of the problems—in fact, they add some of their own—they do make DLLs more usable for the MFC programmer.

I simply can't pass up a challenge, so I followed Ruediger's advice and put **CGL** into a DLL called GLlib.DLL. (The debug version is GLlib-d.DLL.) Placing **CGL** in a DLL didn't solve many problems. The best feature of GLlib.DLL is that I can make changes to the DLL without having to recompile all the applications that use it. Now, this works only if I don't change the DLL interface. The worst feature of GLlib.DLL is that it has to be in the path or directory of any application that calls it. I will tell you more about **CGL** when I build more sample applications that use it.

AFX_EXT_CLASS

Making MFC extension DLLs is very easy with Visual C++ version 2.0, because AppWizard can now create the framework for a DLL. Exporting classes from the DLL is very simple—just add AFX_EXT_CLASS to the class definition:

```
class AFX_EXT_CLASS CGLView : public CView
{
```

```
.
.
.
};
```

In situations such as the above, where a non-exported class is used as the base class for an exported class, Visual C++ generates the following warning message:

```
warning C4275: non dll-interface class 'CView' used as base
for dll-interface class 'CGLView'
```

If the client application called a member function in **CGLView** that was inherited from **CView**, the function would not be found because it was not exported. This warning can be ignored because the client application and GLlib share the same version of **CView** in the MFC DLL. If a client application calls a **CView** function via **CGLView**, it will link to the function in the shared MFC DLL.

For AFX_EXT_CLASS to work, the _AFXEXT preprocessor definition must be defined. AppWizard adds the _AFXDLL preprocessor definition by default, so I removed _AFXDLL from the link line and added _AFXEXT in its place.

Using CGL

In this section, I've listed the steps required to use **CGL**.

The following steps involve **CGL** code directly:

1. Derive a class from **CGL**.
2. Implement the **OnInit**, **OnResize**, and **OnRender** member functions.

The next set of steps involve the application framework:

1. Build the application framework with AppWizard.
2. Add the OpenGL include files (GL\GL.H and GL\GLU.H) to STDAFX.H.
3. Link with the OpenGL library files (OPENGL32.LIB and GLU32.LIB).
4. Link with the **CGL** library file (GLlib.LIB or GLlib-d.DLL).
5. Add an instance of the class you derived from **CGL** to the members in your view class.
6. Forward palette messages from the frame to the active view. (See "OpenGL II: Windows Palettes in RGBA Mode" in the MSDN Library.)
7. Implement handlers for CView::OnPaletteChanged and CView::OnQueryNewPalette.
8. Implement a handler for CView::OnEraseBkgnd.
9. Implement a handler for CView::PreCreateWindow.
10. Add calls to CGL::Create, CGL::Init, CGL::Resize, and so on, to the application's view class.

As you can see, using **CGL** requires many steps. Most of these steps involve adding **CGL** to the view class. Too bad there isn't some way to connect a class to a view class automatically. MFC works around this problem by defining specialized view classes or by enabling AppWizard to perform the required steps. Maybe in a future version of Visual C++ we'll be able to add our own customized additions to AppWizard. Until then, it would be nice if using **CGL** didn't require quite so many steps.

I decided to add a **CGLView** class to GLlib.DLL to alleviate this problem. You still have to perform steps 1 – 6 and a few more steps we'll talk about in a little bit. However, you don't have to implement so many message handlers.

CGLView

I added the **CGLView** class to GLlib.DLL to simplify the creation of applications that use **CGL**. The application's view class is derived from **CGLView**. Figure 4 shows the hierarchy of a typical application that uses the GLlib.DLL classes.

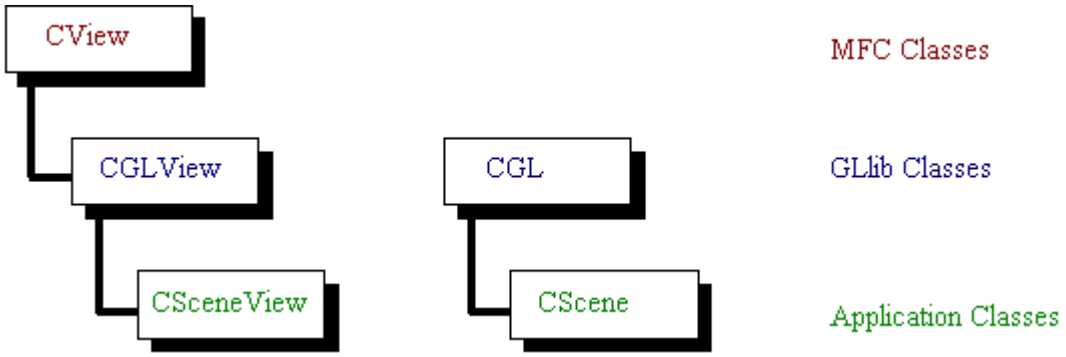

Figure 4. Class hierarchy of an application that uses GLlib

I decided to use **CScene** and **CSceneView** as the names of classes that I inherited from **CGL** and **CGLView**. (You can use other names if you wish.) I expect to use the box, pyramid, and dodecahedron from EasyGL in other sample applications, so I will probably copy the classes over to those applications.

CGLView implements message handlers (for example, **OnSize)**, so they don't have to be implemented in an application class such as **CSceneView**.

```
void CGLView::OnSize(UINT nType, int cx, int cy)
{
   CView::OnSize(nType, cx, cy) ;
   m_pGL->Resize(cx,cy) ;
}
```

Calling **CGL::Resize** through a pointer results in a virtual function call to **OnResize**. If **CScene** is a derived class of **CGL**, the pointer **m_pGL** must point to a **CScene** type. **CGLView** needs to get a pointer to the **CScene** object from **CSceneView**. The pure virtual function **CGLView::GetGLptr** performs this function:

```
class CGLView : public CView
{
.
.
.
protected:
   virtual CGL* GetGLptr() = 0 ;
   CGL* m_pGL ;
.
.
.
};
```

Because **GetGLptr** is a pure virtual function, it must be implemented in a derived class of **CGLView**:

```
class CSceneView : public CGLView
{
.
.
.
protected:
   CScene aScene ;
   virtual CGL* GetGLptr() {return &aScene;}
.
.
.
}
```

CGLView calls **GetGLptr** when handling the WM_CREATE message:

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

   m_pGL = GetGLptr() ;
   BOOL bResult = m_pGL->Create(this) ;
   if (bResult)
      return 0 ;
   else
      return -1;
}
```

DYNCREATE and Pure Virtual Functions

Pure virtual functions are nice because the compiler will warn you if you don't implement them. However, you can't create instances of a class that contains pure virtual functions. Classes with pure virtual functions will generate errors if they use the **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE** macros, because these macros create functions that create instances of the class, which is not allowed with pure virtual functions.

The workaround is easy: Don't include **DECLARE_DYNCREATE** and **IMPLEMENT_DYNCREATE** in **CGLView**. If you look at GLVIEW.H and GLVIEW.CPP in the GLlib.DLL, you will see that these macros are commented out. We will need to add these macros to the classes derived from **CGLView**. For example, if we derive **CSceneView** from **CGLView**, we will add these macros to **CSceneView**. **CSceneView** passes **CView** instead of **CGLView** to **IMPLEMENT_DYNCREATE**, skipping the parent class that does not include **IMPLEMENT_DYNCREATE**. **CSceneView** includes the line:

```
IMPLEMENT_DYNCREATE(CSceneView, CView)
```

instead of:

```
IMPLEMENT_DYNCREATE(CSceneView, CGLView)
```

For more information, see the Knowledge Base article Q103983, "INF: Serializing an Abstract Base Class."

EasyGL, CScene, and CSceneView

The EasyGL sample application is basically the same as GLEasy, except that EasyGL uses GLlib.DLL and GLEasy doesn't. GLlib.DLL contains all the Windows NT OpenGL implementation code, and the **CScene** class contains all the OpenGL code.

CScene

CScene is a simple class: It inherits from **CGL** and implements the **OnResize**, **OnInit**, and **OnRender** member functions. The **OnResize** code is from **CEasyGLView::OnSize**, the **OnInit** code is from **CEasyGLView::PrepareScene**, and the **OnRender** code is from **CEasyGLView::DrawScene**.

CSceneView

CSceneView is a tad more complicated than **CScene**. I let AppWizard build **CSceneView** for me, then modified it. The modifications are pretty simple:

- Include GLLIB.H and SCENE.H in SCENEVW.H.
- Modify **CSceneView** to inherit from **CGLView**.
- Add the following code:

  ```
  CScene aScene ;
  virtual CGL* GetGLptr() { return &aScene; }
  ```

- Pass **CView** to the **IMPLEMENT_DYNCREATE** macro:

  ```
  IMPLEMENT_DYNCREATE(CSceneView, CView /* not CGLView */) ;
  ```

- Comment out **CSceneView::OnDraw**. **CScene::OnRender** and **CGLView::OnDraw** handle the drawing.

By the way, I didn't add the rotation code to EasyGL, but left it as an exercise for the reader. It's pretty easy to do.

Using CGL and CGLView

In this section, I will explain how you can use **CGL** and **CGLView** together, starting from scratch. I will build the framework for EasyCI, which uses OpenGL color index mode. See my article "OpenGL IV: Color Index Mode" in the MSDN Library for a discussion of color index mode and the changes I made to **CGL** to support this mode.

- Create a new MFC application using AppWizard.

  I named my project EasyCI and created a single-document interface (SDI) application without open database connectivity (ODBC) or OLE support. I also decided not to include print preview, the status bar, and the toolbar. GLlib is an MFC extension DLL, so make sure that you link to MFC through the shared DLL (MFC30(D).DLL). MFC extension DLLs link to the shared MFC DLL and require any application that calls them to do the same. For more information, see MFC "Technical Note 33: DLL Version of MFC" in the Visual C/C++ Product Documentation section of the MSDN Library.

  I also changed the name of the view class from **CEasyCIView** to **CSceneView** (and almost forgot to rename the file). You don't have to rename the view class; I did because I wanted to copy the **CScene** and **CSceneView** classes from EasyGL to get the box, pyramid, and dodecahedron.

- Add the following .LIB files to the link settings:

```
GLlib.lib
opengl32.lib
glu32.lib
glaux.lib (optional)
```

- Follow the instructions in the "EasyGL, CScene, and CSceneView" section of this article, or copy **CScene** and **CSceneView** from EasyGL (from the SCENE.CPP, SCENE.H, SCENEVW.CPP, and SCENEVW.H files). You'll have to change the **CEasyglDoc** references to **CEasyCIDoc** in SCENEVW.H and SCENEVW.CPP. Also, change the names of the included files in SCENEVW.CPP. Add **CScene** to the project.

- Forward palette messages from the frame to the active view. (See "OpenGL II: Windows Palettes in RGBA Mode" in the MSDN Library.)

- Ignore the two C4275 errors that you get when you compile. See the section "CGL in a DLL" earlier in this article for more information about this error message.

- Make sure that GLlib(-d).DLL is on the path before you try to execute your new program.

That's all there is to it!

Conclusion

CGL is a simple, usable class library for OpenGL. **CGL** proves that you can build a small C++ class that simplifies the use of OpenGL without changing the OpenGL code itself. Look for **CGL** to grow as I extend it for my future articles on OpenGL.

Bibliography

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

Microsoft Knowledge Base Q103983. "INF: Serializing an Abstract Base Class." (MSDN Library, Knowledge Base)

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 I: Quick Start.". December 1994. (MSDN Library, Technical Articles)

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

Microsoft Win32 Software Development Kit (SDK) for Windows NT 3.5 *OpenGL Programmer's Reference*.

"Technical Note 33: DLL Version of MFC." (MSDN Library, Developer Products, Visual C/C++ Microsoft Foundation Class Reference, MFC Notes).