3. OpenGL Under MFC

Seven basic steps are required to use OpenGL calls in any Win/NT program:

1. Initialize the client area of a window
2. Get a Device Context (DC) for the rendering location
3. Select and set a pixel format for the device context
4. Create a Rendering Context (RC) associated with the DC
5. Draw using OpenGL Commands
6. Delete the RC
7. Release the DC

- **Windows Styles**

  OpenGL can render only into the client area of a window that has been initialized for it

  The style of an OpenGL window should include

  ◊ **WS_CLIPSIBLINGS**

  ◊ **WS_CLIPCHILDREN**

  - This prevents OpenGL from attempting to draw into other windows

  - The style is an argument to `CreateWindow`
- The style may not include CS_PARENTDC
- A pixel format may not be set for a window without these style bits set

Style bits are normally set by overriding `PreCreateWindow`

**Example 3.1**

```cpp
BOOL COpenGLView::PreCreateWindow(CREATSTRUCT& cs)
{
    cs.style |= WS_OVERLAPPED |
                WS_CLIPCHILDREN |
                WS_CLIPSIBLINGS;
    return CView::PreCreateWindow(cs);
}
```

**Device Contexts**

A Device Context (DC) is a Windows data structure containing information about the drawing attributes of a device

- Each window has a device context to receive the graphics output
- Device contexts allow device-independent drawing in Windows
- Can be used to draw to the screen, to the printer, or to a metafile

The DC is used to create an OpenGL rendering context
- This rendering context is used by OpenGL to draw to the DC and ultimately the device surface

Normally, the DC is created during initialization and destroyed as the application closes

- Normal windows programs release the DC following each repainting of the window

- Animated OpenGL programs generally hold onto their DC throughout execution

(Not possible under 16-bit windows due to limited number of DC’s)

Example 3.2
Creating a DC

\[
m_pDC = \text{new CClientDC(this)};
\]

♦ GDI

GDI is the original Windows 2D graphics interface.

All GDI calls pass through a DC

- All GDI functions include the device context as the first parameter

- GDI functions cannot be used when the window is enabled for OpenGL double buffering
• **Pixel Formats**

Pixel formats are an extension of the Win32 API to support OpenGL.

A pixel format specifies properties of a drawing surface involving the organization and layout of color bits.

The pixel format sets a device context’s OpenGL properties such as:

- Single or double buffering
- RGBA or color indexing
- Drawing to a window or bitmap
- Support of GDI or OpenGL calls
- Color depth
- Z buffer depth
- stencil, accumulation, aux buffers?

Once the pixel format for a window has been set, it can never be reset.
Generic implementation of OpenGL 1.1 and Windows 95/NT supports 24 pixel formats

- Installed hardware may change or add capabilities
- Each format is referred to by a number
- These numbers may change
- To retrieve the maximum number of device pixel formats supported for a given DC, use the DescribePixelFormat function

♦ Setting Up the Pixel Format

Setting a pixel format for a DC is a three step process

1. Fill out the PIXELFORMATDESCRIPTOR structure according to the desired window characteristics

2. Obtain the id of the closest available format by passing the structure using ChoosePixelFormat

3. Set the pixel format for the DC using the SetPixelFormat function
Example 3.3

Setting a pixel format

```cpp
BOOL COpenGLView::SetupPixelFormat()
{
    static PIXELFORMATDESCRIPTOR pfd =
    {
        sizeof( PIXELFORMATDESCRIPTOR ), // size of pfd
        1,                     // version number
        PFD_DRAW_TO_WINDOW |   // support window
            PFD_SUPPORT_OPENGL |  // support OpenGL
            PFD_DOUBLEBUFFER,     // double buffered
        PFD_TYPE_RGBA,         // RGBA type
        24,                    // 24-bit color
        0, 0, 0, 0, 0, 0,      // color bits ignored
        0,                     // no alpha buffer
        0,                     // shift bit ignored
        0,                     // no accumulation buffer
        0, 0, 0, 0,            // accum bits ignored
        32,                    // 32-bit z-buffer
        16, // better performance with 16-bit z-buffer
        0,                     // no stencil buffer
        0,                     // no auxiliary buffer
        PFD_MAIN_PLANE,        // main layer
        0,                     // reserved
        0, 0, 0                // layer masks ignored
    };

    int pixelformat;
    pixelformat =
        ChoosePixelFormat( m_pDC->GetSafeHdc(), &pfd );

    SetPixelFormat( m_pDC->GetSafeHdc(),
        pixelformat, &pfd );

    return TRUE;
}
```
• **OpenGL Rendering Contexts**

Under Windows, the OpenGL environment is embodied in the Rendering Context (RC)

Extension to the Windows API to support OpenGL

- A RC is bound to a DC which is bound to a “window”

- An RC in an OpenGL program plays the same role that a DC plays in a GDI

- Only one RC may be current per thread

- The pixel format must be set up before creating the RC

- The RC must be associated with a DC before OpenGL commands can be executed

- The DC should not be released or deleted when it is associated with a RC

Three WGL functions are used to manage RC’s

```
HGLRC wglCreateContext( HDC hDC );

BOOL wglDeleteContext( HGLRC hrc );

BOOL wglMakeCurrent( HDC hDC, HGLRC hrc );
```
- **wglCreateContext** takes a handle to a windows GDI DC and creates an RC

- **wglDeleteContext** is used to *delete* an RC after use

- **wglMakeCurrent** is used to make a specified RC current for a DC

  ◊ The DC must have the same characteristics as the DC under which the RC was created

  ◊ In particular the characteristics embodied by the pixel format must be the same

RC’s are generally created and deleted in the WM_CREATE and WM_DESTROY message handlers respectively

Additional WGL functions for RC management include **wglGetCurrentContext** and **wglGetCurrentDC**

**Example 3.4**

Creating a RC

```c
m_hRC = wglCreateContext( hDC );
```

**Example 3.5**

Deleting a RC

```c
wglDeleteContext( m_hRC );
```