Abstract
The PFD_DRAW_TO_BITMAP pixel format descriptor flag allows OpenGL™ applications to render on a Microsoft® Windows® device-independent bitmap (DIB). The resulting DIB can be manipulated to the full extent using the commands in the Windows graphics device interface (GDI). This article explains how you can render OpenGL scenes on DIBs with PFD_DRAW_TO_BITMAP. The EasyBit sample application demonstrates the techniques presented in the article.

Introduction
Recently, I took a trip back home to visit my parents. The first evening, I walked out to the dock to watch the setting sun turn the sky crimson orange. The silence of the coming night didn’t match the intensity of the fiery inferno appearing in the sky. Eventually, the fire subsided into the waters of the lake, and the only lights were those of the stars above.

I lay down on the dock to see the stars better. It wasn't long before I felt something pushing tentatively on my chest. Satisfied with his search, the neighbor's cat, Zephyr, climbed on my chest, purring with an intensity rivaling an asthma attack. I was amazed. I had only known the cat for nine months, and I had last seen him over two years ago. Did he remember me? Did he even know who I was? Did he care?

Like Zephyr, who doesn't care whose lap he sits on, OpenGL™ doesn't care which surface it renders on. Again like Zephyr, who pokes my chest to see whether it's okay to lie down on it, OpenGL must make sure that the rendering surface is okay. Currently, OpenGL will render in windows and on bitmaps. Zephyr is more promiscuous than that; I have no idea how many laps he sits on.

The previous articles in this series focused on rendering in a window. This article will focus on rendering on a bitmap or, more specifically, on a device-independent bitmap (DIB) section. The article covers the following topics:

- A description of the EasyBit sample application
- Reasons for drawing on bitmaps
- Using the PFD_DRAW_TO_BITMAP pixel format descriptor
- Modifications to GLlib to support rendering on DIBs
- A discussion of DIB sections
- Selecting the correct pixel format
- Complications introduced by palettes
- Using the same palette as the system to improve performance
- A guide to troubleshooting your application

EasyBit Sample Application
The EasyBit sample application demonstrates how to render OpenGL images on a bitmap. Figure 1 below provides a simulated screen shot of EasyBit. (I didn't use a real screen shot because the Microsoft® Development Library software doesn't display 256-color images.)
Figure 1. Simulated screen shot from EasyBit

The client area of the window is rendered using the same code (CScene) that EasyGL uses. This scene is rendered on the window (PFD_DRAW_TO_WINDOW) using double buffering (PFD_DOUBLEBUFFER). As you can see in Figure 1, the left side of the screen contains four squares containing images rendered by OpenGL. In the first square, CScene renders on an 8-bits-per-pixel (bpp) CDIBSurface, a class defined in Nigel Thompson's animation library. CDIBSurface was created after Nigel's book, Animation Techniques for Win32, was released.)

In the next three squares, CSceneBox, CScenePyramid, and CSceneDodec render on an 8-bpp, 16-bpp, and 24-bpp CSimpleDIB, which is defined in EasyBit. Both CSimpleDIB and CDIBSurface will be discussed later in this article. The graphics device interface (GDI) Rectangle command draws a border around the DIBs, and TextOut writes a caption for each DIB. As you can see, GDI and OpenGL commands can be mixed on the DIB.

I added the ability to rotate any object (from the GLEasy sample application) to EasyBit. The user selects a shape from the Rotate menu, which is the self-drawing menu illustrated in Figure 2.

Figure 2. Self-drawing menu in EasyBit

CSceneBox, CScenePyramid, and CSceneDodec are rendered on CSimpleDIBs for display in the Rotate menu. If the screen has 8 bpp, 8-bpp CSimpleDIBs are created for rendering the OpenGL scenes. If the screen does not have 8 bpp, 24-bpp CSimpleDIBs are created. (See the article "MFC Self-Drawing Menus" in the Development Library for more information on how I created this menu.)

The shape currently selected for rotating is displayed in the status bar, as shown in Figure 3. For more information on customizing the status bar, see my article "Bitmaps and Other CStatusBar Customizations" in the Development Library.

Figure 3. EasyBit status bar with bitmap of shape to rotate

EasyBit also measures the blting performances of the DIBs with different bpp; select the Blting option from the Time menu.

When a shape is rotating, I decided not to display the figures on the left side of the EasyBit screen (see Figure 1), because displaying these bitmaps resulted in too much flickering. The main client area is painted using CScene with double buffering. The new GDI SwapBuffers function in Win32® copies the contents of the back buffer to the screen. It paints the entire client area of the window.
associated with the current rendering context. This requires blitting the bitmaps on the left side of the screen each time the shape rotates.

If you wish to display the bitmaps but avoid flashing, you can use one of the following options:

- Create a child window on the view on which CScene rendered. SwapBuffers can update the client area of the child window instead of the client area of the view.
- Change CScene to render on a bitmap, and then blt this bitmap to the screen.

However, I didn't bother because I didn't think it was necessary to display the bitmaps on the screen while the object was rotating.

Class Map

The table below describes the important classes in EasyBit.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shapes.cpp</td>
<td>&lt;no class&gt;</td>
<td>Contains code to build OpenGL display lists for the box, pyramid, and dodecahedron.</td>
</tr>
<tr>
<td>scene.cpp</td>
<td>CScene</td>
<td>Identical to the CScene class in EasyGL, except that it uses the display lists in shapes.cpp and includes rotation for the shapes.</td>
</tr>
<tr>
<td>scenevw.cpp</td>
<td>CSceneVw</td>
<td>Inherits from the GLlib CGLView class. I also added rotation support and code to render images on the DIB sections displayed on the left side of the EasyBit screen.</td>
</tr>
<tr>
<td>csimpledib.cpp</td>
<td>CSimpleDIB</td>
<td>Implements a simple encapsulation of a DIB section. Supports the creation of 8-, 16-, and 24-bpp DIB sections. OpenGL images are rendered on these DIB sections. The application also uses Nigel Thompson’s CDIBSurface class.</td>
</tr>
<tr>
<td>cscenebox.cpp</td>
<td>CSceneBox</td>
<td>Inherits from the GLlib CGL class. Uses the OpenGL display list for a box contained in shapes.cpp.</td>
</tr>
<tr>
<td>cscenepyramid.cpp</td>
<td>CScenePyramid</td>
<td>Inherits from the GLlib CGL class. Uses the OpenGL display list for a pyramid contained in shapes.cpp.</td>
</tr>
<tr>
<td>cscenedodec.cpp</td>
<td>CSceneDodec</td>
<td>Inherits from the GLlib CGL class. Uses the OpenGL display list for a dodecahedron contained in shapes.cpp.</td>
</tr>
<tr>
<td>sceneci.cpp</td>
<td>CSceneCI</td>
<td>A color index mode version of CScene. Demonstrates the creation of an identity palette. The code for CSceneCI comes from the EasyCI sample application.</td>
</tr>
<tr>
<td>cmystatusbar.cpp</td>
<td>CMyStatusBar</td>
<td>Implements a customized status bar. See &quot;Bitmaps and Other CStatusBar Customizations&quot; in the Development Library and Figure 3 earlier in this article.</td>
</tr>
<tr>
<td>cownernmenu.cpp</td>
<td>COwnerMenu</td>
<td>Implements the self-drawing Rotate Menu. See &quot;MFC Self-Drawing Menus&quot; in the Development Library and Figure 2 earlier in this article.</td>
</tr>
</tbody>
</table>

Reasons for Drawing on Bitmaps

In the previous articles in this series, we covered rendering OpenGL directly to the client area of a window, as illustrated in Figure 4.

Figure 4. Rendering in a window

In "OpenGL I: Quick Start" and "Windows NT OpenGL: Getting Started", we discussed how to set up the pixel format descriptor of a window so we could use OpenGL commands. Rendering to a window requires using the PFD_DRAW_TO_WINDOW flag in the pixel format descriptor.

In some cases, you don't want to draw directly to the screen. Nothing looks worse than watching an application paint the client area piece by piece. For smooth animation, off-screen rendering isn't only nice, it's required. OpenGL for Windows NT™ supports off-screen
rendering whenever the pixel format for a window is set for double buffering. Use the PFD_DOUBLEBUFFER pixel format descriptor flag to enable double buffering for a particular window. My OpenGL sample applications use double buffering. With double buffering enabled, OpenGL renders to an off-screen buffer, and the GDI SwapBuffers command moves the contents of the off-screen buffer onto the screen. Figure 5 illustrates this process.

Figure 5. Double buffering

For the generic pixel formats, the off-screen buffer is a DIB section. Some OpenGL graphic accelerators use special hardware to implement the off-screen buffer on the card.

Double buffering, as supported by the Windows NT implementation of OpenGL, solves many of the problems caused by rendering directly to the screen. However, you cannot access the off-screen buffer. You can't draw on it or manipulate it, except with OpenGL commands.

Another limitation is caused by the SwapBuffers command. When SwapBuffers is called, it clears the entire client area of the window associated with the current rendering context. There is no way to restrict SwapBuffers to a specific part of the client area. Remember that SwapBuffers is a Windows® GDI command and not part of OpenGL. The OpenGL scissor and stencil features are generally used to restrict OpenGL from drawing to parts of the screen, but they do not affect SwapBuffers. This can be a major limitation if you are mixing GDI and OpenGL in your application. Another limitation of SwapBuffers is that the effects of calling it a second time are undefined. Thus, you can't render a scene once, and then call SwapBuffers multiple times to update the display.

An approach to off-screen rendering that provides the application with more control is rendering on a bitmap. If you use the PFD_DRAW_TO_BITMAP pixel format descriptor flag instead of PFD_DRAW_TO_WINDOW, OpenGL will render the image on a bitmap instead of rendering it to a window. After OpenGL has rendered on the bitmap, you can use GDI commands to do whatever you want with the bitmap, from simply blitting it on the screen to writing text on the bitmap. If you use a DIB section (more on this later), you can even play with the bits of the bitmap directly. Figure 6 illustrates how OpenGL renders on a bitmap.

Figure 6. Rendering on a bitmap

Rendering images on a bitmap gives the programmer much better control over the display of the final image.

**PFD_DRAW_TO_BITMAP**

OpenGL will render on anything that has a proper pixel format descriptor. (Dr. GUI is working on a pixel format descriptor to allow rendering on lattes, since he spends so much time looking at his coffee.) Setting the pixel format descriptor is always the first step. When rendering on a window, we set the pixel format descriptor for a device context (DC) associated with the window. When rendering on a bitmap, we set the pixel format descriptor for the DC containing the selected bitmap.

After setting the pixel format to PFD_DRAW_TO_BITMAP, we must use the same DC for all wglMakeCurrent calls that we used in the SetPixelFormat and wglCreateContext calls. This differs from PFD_DRAW_TO_WINDOW, where we can use any DC associated with the window on which we are drawing. This means that we cannot "unselect" the bitmap in the DC and replace it with another bitmap. It also means that we cannot resize the bitmap without starting over and resetting the pixel format.

You can use PFD_DRAW_TO_BITMAP with either PFD_TYPE_RGBA or PFD_TYPE_COLORINDEX; you cannot use it with PFD_DOUBLEBUFFER.

The following code sets up a PIXELFORMATDESCRIPTOR structure for rendering on a bitmap:

```c
PIXELFORMATDESCRIPTOR pfd;
memset(&pfd,0,sizeof(PIXELFORMATDESCRIPTOR));
pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion = 1;
pfd.dwFlags = PFD_DRAW_TO_BITMAP;// replaces PFD_DRAW_TO_WINDOW
   | PFD_SUPPORT_OPENGL |
   | PFD_SUPPORT_GDI;
pfd.iPixelType = PFD_TYPE_RGBA;
```
When filling in the pixel format descriptor, you must set the `cColorBits` element to the number of bits per pixel in the bitmap. ChoosePixelFormat gives you the number of bits per pixel you asked for, not the number of bits per pixel for the bitmap format. (Note that the behavior of ChoosePixelFormat changes, depending on the pixel format descriptor flag. If you use PFD_DRAW_TO_WINDOW, ChoosePixelFormat returns an appropriate `cColorBits` value based on the screen resolution.)

**Changes to GLlib**

I had to make some changes to GLlib to support rendering on bitmaps. GLlib doesn’t contain code that is specific to DIB sections, bitmaps, or CDIBSurface. The only changes I made were in the `CGL` class; I left `CGLView` and `CGLImage` unchanged.

**CGL::Create**

The main change to `CGL::Create` for bitmap-rendering support is a new function that takes a `CDC*` instead of a `CWnd*`. The new function prototype is listed below:

```c
BOOL Create(CDC* pdcMemory,
           int iPixelType = PFD_TYPE_RGBA,
           DWORD dwFlags  = PFD_SUPPORT_OPENGL |  // Use OpenGL.
PFD_SUPPORT_GDI |
PFD_DRAW_TO_BITMAP ); // Pixel format for bitmap
```

Notice the difference in default values between the new bitmap-rendering `Create` function and the existing window-rendering `Create` function:

```c
BOOL Create(CWnd* pWnd,
            int iPixelType  = PFD_TYPE_RGBA,
            DWORD dwFlags   = PFD_DOUBLEBUFFER |    // Use double buffer.
PFD_SUPPORT_OPENGL |  // Use OpenGL.
PFD_DRAW_TO_WINDOW ); // Pixel format for window
```

The code for the new `Create` function is shown below. The important changes are in bold.

```c
BOOL CGL::Create(CDC* pdcMemory, int iPixelType, DWORD dwFlags)
{
    m_pdc = pdcMemory ;
    CBitmap* pBitmap = m_pdc->GetCurrentBitmap() ;
    BITMAP bmInfo ;
    pBitmap->GetObject(sizeof(BITMAP), &bmInfo) ;
    ASSERT(bmInfo.bmPlanes == 1) ;
    ASSERT((bmInfo.bmBitsPixel == 8) ||
           (bmInfo.bmBitsPixel == 16) ||
           (bmInfo.bmBitsPixel == 24)) ;
    // Fill in the pixel format descriptor.
    PIXELFORMATDESCRIPTOR pfd ;
    memset(&pfd,0,sizeof(PIXELFORMATDESCRIPTOR)) ;
    pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
    pfd.nVersion = 1 ;                           // Version number
    pfd.dwFlags = dwFlags ;
    pfd.iPixelType = iPixelType ;
    pfd.cColorBits = (BYTE)bmInfo.bmBitsPixel ;
    pfd.cDepthBits = 32 ;                        // 32-bit depth buffer
    pfd.iLayerType = PFD_MAIN_PLANE ;           // Layer type
    // Let children change creation.
    OnCreate(NULL, &pfd) ;
    ASSERT( (dwFlags & PFD_DRAW_TO_BITMAP)) ;
    ASSERT( !(dwFlags & PFD_DOUBLEBUFFER)) ;
    ASSERT( (iPixelType == PFD_TYPE_RGBA) ||
            (iPixelType == PFD_TYPE_COLORINDEX)) ;
    // Draw onto a bitmap.
    m_bDrawToBitmap = TRUE ;
}
// Determine double buffering state.

m_bDoubleBuffer = FALSE;

// Choose the pixel format.
int nPixelFormat = ChoosePixelFormat(m_pdc->m_hDC, &pfd);
if (nPixelFormat == 0)
{
    TRACE("ChoosePixelFormat Failed %d\n", GetLastError());
    return FALSE;
}
TRACE("Pixel Format %d\n", nPixelFormat);

// Set the pixel format.
BOOL bResult = SetPixelFormat(m_pdc->m_hDC, nPixelFormat, &pfd);
if (!bResult)
{
    TRACE("SetPixelFormat Failed %d\n", GetLastError());
    return FALSE;
}

// Create the palette.
CreatePalette();

// Create a rendering context.

m_hrc = wglCreateContext(m_pdc->m_hDC);
if (!m_hrc)
{
    TRACE("wglCreateContext Failed %x\n", GetLastError());
    return FALSE;
}
return TRUE;

As you can see from the code above, we pass a CDC pointer instead of a CWnd pointer to the Create function. The DC should have selected the bitmap, DIB, or DIB section to be rendered on. The number of bits per pixel is determined by a call to CDC::GetObject. The Boolean m_bDrawToBitmap is set to TRUE. The rest of the code is unchanged.

**CGL::CreatePalette**

I also had to change CGL::CreatePalette to support 16- and 24-bpp displays. Otherwise, if the display is 24 bpp and you try to render on an 8-bpp DIB, PFD_NEED_PALETTE is not set and CGL::CreatePalette will not create a palette.

To fix this problem, I changed CGL::CreatePalette to create a palette if m_bDrawToBitmap is TRUE and the number of bits per pixel in the pixel format descriptor is 8. The code is shown below.

```cpp
BOOL b8bppDIB = (m_bDrawToBitmap) && (pfd.cColorBits == 8) ;
if ( (pfd.dwFlags & PFD_NEED_PALETTE) ||
    (b8bppDIB) ||
    (bColorIndex))
{

}
```

**m_bDrawToBitmap**

In addition to these major changes, I made a number of small modifications to the GLlib code. Most of these changes use the Boolean m_bDrawToBitmap. The CGL::Create functions call CGL::CreatePalette, which normally selects a palette into the DC associated with a rendering context. We do not want to do this with DIB sections, so I changed CGL::CreatePalette as shown below.

```cpp
BOOL CGL::CreatePalette()
{

// Select and realize palette.
if ( (m_bDrawToBitmap) // BIT
{
    m_pOldPal = m_pdc->SelectPalette(m_pPal, 0);
    m_pdc->RealizePalette();
}
```
We don't own the DC given to us at creation time, so we shouldn't delete it:

```cpp
void CGL::Destroy()
{
  // ...

  if (m_pdc && !m_bDrawToBitmap)
  {
    delete m_pdc ;
    m_pdc = NULL ;
  }
}
```

Those are the changes I had to make to GLlib to handle rendering on bitmaps. In some of my earlier designs, I handled the DIBs and bitmaps in the `CGL` class itself. After I used the class, it quickly became apparent that the DIB code did not belong in `CGL`. Having `CGL` depend only on a `CDC` instead of an internal or external DIB or bitmap class makes it more useful, as the code in the next section demonstrates.

### DIB Sections

The EasyBit sample application works exclusively with DIB sections. A DIB section is a special version of a DIB that was introduced in Windows NT version 3.5 and is supported in Windows 95. DIB sections have two advantages over normal DIBs and bitmaps:

- You can draw on DIB sections with GDI, and you can directly modify the bits in memory. This differs from device dependent bitmaps (on which only GDI can draw), and DIBs (which don't support GDI but which you can modify directly).
- DIB sections (especially large DIBs) can be blted to the screen faster than normal DIBs and bitmaps.

For more information on DIB sections, see Nigel Thompson's book *Animation Techniques for Win32* (available from Microsoft Press®) and latest animation articles.

### CDIBSurface

Speaking of Nigel, I rendered OpenGL scenes on his `CDIBSurface` class. `CDIBSurface` encapsulates both a DIB section and a DC within one object that is easy to manipulate. The scene in the upper-left corner of the client area in Figure 1 is a `CScene` object rendered on a `CDIBSurface` object.

I used the following `CDIBSurface` commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Creates a <code>CDIBSurface</code> object. You can specify the size and a palette.</td>
</tr>
<tr>
<td>GetDC</td>
<td>Gets the DC attached to the <code>CDIBSurface</code>.</td>
</tr>
<tr>
<td>SetPalette</td>
<td>Sets the palette for the DIB section.</td>
</tr>
<tr>
<td>BitBlt</td>
<td>Blts the DIB section to the screen. Make sure that you are using the latest version of the animation library; some early versions of <code>CDIBSurface</code> selected the palette into the destination DC before blting.</td>
</tr>
<tr>
<td>Draw</td>
<td>Draws the <code>CDIBSurface</code> object by calling <code>BitBlt</code>.</td>
</tr>
</tbody>
</table>


To render on a `CDIBSurface` object, you must first create the object. The code below, taken from `CSceneView::OnSize`, creates a `CDIBSurface` object with no palette. The `sizeBitmap` parameter contains the size of the `CDIBSurface` object. The last parameter is a pointer to a `CPalette` object. I don't have a palette built yet, so I pass in a NULL.

```cpp
CDIBSurface aDIBSurface ;
aDIBSurface.Create(sizeBitmap.cx, sizeBitmap.cy, NULL) ;
```

I need the DC to create the OpenGL rendering context, so I get the DC for the `CDIBSurface` object:

```cpp
CDC* pdcTemp = aDIBSurface.GetDC() ;
```

I pass the DC to the `CScene` object's `Create` function. `CScene` inherits from `CGL`, which is contained in GLib. `CGL` has two `Create` functions, one of which takes a pointer to a `CDC` object. This is the `Create` function you should use when you want to render on a bitmap:
When I created the CDIBSurface object, I didn't specify a palette because the palette hadn't been constructed yet. I need a DC before I can create CScene, which creates the palette. Therefore, I get a pointer to the palette used in aSceneOnDib to set the palette in the aDIBSurface.

CPalette* pPalTemp = aSceneOnDib.GetPalette() ;
if (pPalTemp) aDIBSurface.SetPalette(pPalTemp) ;

CDIBSurface::SetPalette does not copy the palette that pPalTemp points to, so you should keep the palette around for the life of aDIBSurface. This is not a requirement of a DIB section, but of the CDIBSurface class. Later, we will look at CSimpleDIB::SetPalette to see how CDIBSurface::SetPalette works.

Now we can actually do the rendering:

aSceneOnDib.Resize(sizeBitmap.cx, sizeBitmap.cy) ; //Doesn't resize?!!!!
aSceneOnDib.Init() ;
aSceneOnDib.Render() ;

The code above calls the CScene::Resize, CScene::Init, and CScene::Render functions, which are discussed in the "OpenGL III: Building an OpenGL C++ Class" article in the MSDN Library.

CScene::Resize does not resize the CDIBSurface object; it only sets up the projection matrices for transforming a 3-D object onto the 2-D screen.

Finally, we draw a blue border around the bitmap using GDI commands with the CSceneView::DrawBlueBorder helper function:

DrawBlueBorder(pdcTemp) ;

At this point, I could delete the CScene object, aSceneOnDib, except that aDIBSurface contains a pointer to the palette in aSceneOnDib. There are several methods you could use to delete aSceneOnDib. The simplest is to make a copy of the palette object in aSceneOnDib and pass the pointer to the copy to aDIBSurface. You could then delete aSceneOnDib and keep the palette object.

In the EasyBit sample application, there is another way around the problem. The main client area of the window is painted using a CScene object, m_aScene, which has the same palette as the aSceneOnDib object. Therefore, we could simply use the palette from m_aScene. If you have a 16- or 24-bpp display, m_aScene won't have a palette, but you won't need one when drawing an 8-bpp DIB to a 16- or 24-bpp display.

CDIBSurface internals

So far, I've kept this discussion at a fairly high level; I haven't gone into the details of how the CDIBSurface class actually works. Refer to Nigel's book, his technical article "Creating Programs Without a Standard Windows User Interface Using Visual C++ and MFC," and the CDIBSurface source code for more information on the internals of CDIBSurface.

However, I won't leave you up a creek without a paddle, so I will explain the internals of CDIBSurface indirectly. Instead of looking at CDIBSurface internals, I will examine the internals of an equivalent class, CSimpleDIB, which is a minimal encapsulation of a DIB section and DC. For our purposes in this article, CSimpleDIB is equivalent to CDIBSurface but it is much simpler to understand. The next two sections discuss CSimpleDIB.

CSimpleDIB

When I started writing this article, my intention was to use Nigel's CDIBSurface class. However, in the process of trying to use CDIBSurface, I encountered problems that I couldn't debug. The code started to get very confusing: CDIBSurface inherits from CDIB and the creation code builds on top of a DIB object, and it becomes very difficult to see who is controlling what. There is also a lot of WinG-dependent code in CDIBSurface, which only adds to the confusion. I needed to generate 16- and 24-bpp DIB sections in addition to the 8-bpp DIB sections that CDIBSurface creates. I just didn't have the time to add this support to CDIBSurface while ensuring that other CDIBSurface features were not affected.

I thought it would much easier to work with a simpler class first, and then figure out what I needed to do to make the code work with CDIBSurface. I decided that if I found it difficult to understand the internals of CDIBSurface, so would other developers. For these reasons, I decided to write CSimpleDIB.

Using CSimpleDIB

The code below, taken from CSceneView::OnSize, provides an example of how to use CSimpleDIB:

CSimpleDIB m_theSimpleDIB;
m_theSimpleDIB.Create(m_sizeBitmap.cx, m_sizeBitmap.cy, 8) ;
CDC* pdcTemp = m_theSimpleDIB.GetDC() ;

CSceneDodec aSceneDodec;
aSceneDodec.Create(pdcTemp) ;
CPalette* pPalTemp = aSceneDodec.GetPalette();
if (pPalTemp) m_theSimpleDIB.SetPalette(pPalTemp);
aSceneDodec.Resize(m_sizeBitmap.cx, m_sizeBitmap.cy);
aSceneDodec.Init();
aSceneDodec.Render();

DrawBlueBorder(pdcTemp);
DrawCaption(pdcTemp, _T("8 bpp"));

As you can see, **CSimpleDIB** is used in basically the same way as **CDIBSurface**. The only difference is the call to **CSimpleDIB::Create** in the code above. The last parameter to **CSimpleDIB::Create** specifies the number of bits per pixel for the DIB section, whereas **CDIBSurface::Create** provides a pointer to a palette. **CSimpleDIB** supports 8-, 16-, and 24-bpp DIB sections. In the example above, I created an 8-bpp DIB section. The following code creates a 24-bpp DIB section:

m_theSimpleDIB.Create(m_sizeBitmap.cx, m_sizeBitmap.cy, 24);

CDC* pdcTemp = m_theSimpleDIB.GetDC();

CScenePyramid aScenePyramid;
aScenePyramid.Create(pdcTemp);
aScenePyramid.Resize(m_sizeBitmap.cx, m_sizeBitmap.cy);
aScenePyramid.Init();
aScenePyramid.Render();

DrawBlueBorder(pdcTemp);
DrawCaption(pdcTemp, _T("24 bpp"));

Careful readers will have noticed that the following two lines:

CPalette* pPalTemp = aScenePyramid.GetPalette();
if (pPalTemp) m_theSimpleDIB.SetPalette(pPalTemp);

are missing from the example above. **aScenePyramid** is attached to a 24-bpp DIB; therefore, it does not have a palette, so **pPalTemp** will always be NULL.

**CSimpleDIB internals**

**CSimpleDIB** is a bare-bones class that encapsulates both **CreateDIBSection** and a **CDC** object.

class CSimpleDIB
{
    // ...

    public:
    CSimpleDIB();
    virtual ~CSimpleDIB();

    void Create(int cx, int cy, int ibitcount);
    void Draw(CDC* pdcDest, int x, int y);
    void SetPalette(CPalette* pPal);
    CDC* GetDC() {return m_pdc;}

    // ...

};

As the header file above shows, **CSimpleDIB** is simple. It does not inherit from a base class, so there's nothing to learn except what's already in the code above. I tried to make the interface to **CSimpleDIB** similar to **CDIBSection** to keep down the confusion level.

To create a **CSimpleDIB** object, you simply call **Create** with the desired size and number of bits per pixel for the DIB section.

void CSimpleDIB::Create(int cx, int cy, int ibitcount)
{
    ASSERT((ibitcount == 8) || (ibitcount == 16) || (ibitcount == 24));
    ASSERT(cx > 0);
    ASSERT(cy > 0);

    // Destroy parts of objects if we are recreating it.
    if ((m_pdc != NULL) || (m_hbmp != NULL)) destroy();

    // Save size for drawing later.
    m_sizeDIB.cx = cx;
    m_sizeDIB.cy = cy;

    // Create a BITMAPINFOHEADER structure to describe the DIB.
    BITMAPINFOHEADER BIH;
    int iSize = sizeof(BITMAPINFOHEADER);
    //...
memset(&BIH, 0, iSize);
// Fill in the header info.
BIH.biSize = iSize;
BIH.biWidth = cx;
BIH.biHeight = cy;
BIH.biPlanes = 1;
BIH.biBitCount = ibitcount;
BIH.biCompression = BI_RGB;

// Create a new device context.
m_pdc = new CDC;
m_pdc->CreateCompatibleDC(NULL);

// Create the DIB section.
m_hbmp = CreateDIBSection(m_pdc->GetSafeHdc(),
  (BITMAPINFO*) &BIH,
  DIB_PAL_COLORS,
  &m_pBits,
  NULL,
  0);

ASSERT(m_hbmp);
ASSERT(m_pBits);

// Select the new bitmap into the buffer DC.
if (m_hbmp)
{
  m_hbmOld = (HBITMAP)::SelectObject(m_pdc->GetSafeHdc(),
  m_hbmp);
}
}

Calling CreateDIBSection is as simple as filling out a BITMAPINFO structure, which consists of a BITMAPINFOHEADER structure with color information. Because I don't have a palette when I create the DIB section, I don't need the color information, so I simply use a BITMAPINFOHEADER structure. I create a DC because CreateDIBSection requires one as a parameter. When the new DIB section is created, it is selected into the DC.

To display the CSimpleDIB on the screen, call CSimpleDIB::Draw, which simply encapsulates a call to CDC::BitBlt.

void CSimpleDIB::Draw(CDC* pdcDest, int x, int y)
{
  pdcDest->BitBlt(x, y,
    m_sizeDIB.cx, m_sizeDIB.cy,
    m_pdc,
    0, 0,
    SRCCOPY);
}

If the destination DC, pdcDest, has 8 bpp, you'll need to select and realize a palette before calling CSimpleDIB::Draw.

Speaking of palettes... If you are creating an 8-bpp CSimpleDIB object, you will need to set the palette. Otherwise, when you attempt to draw the object, you'll get a black square. To set the palette, use CSimpleDIB::SetPalette.

void CSimpleDIB::SetPalette(CPalette* pPal)
{
  ASSERT(pPal);

  // Get the colors from the palette.
  int iColors = 0;
pPal->GetObject(sizeof(iColors), &iColors);
  ASSERT(iColors > 0);
  PALETTEENTRY* pPE = new PALETTEENTRY[iColors];
  pPal->GetPaletteEntries(0, iColors, pPE);

  // Build a table of RGBQUADS.
  RGBQUAD* pRGB = new RGBQUAD[iColors];
  ASSERT(pRGB);
  for (int i = 0; i < iColors; i++)
  
  pRGB[i].rgbRed = pPE[i].peRed;
pRGB[i].rgbGreen = pPE[i].peGreen;
pRGB[i].rgbBlue = pPE[i].peBlue;
pRGB[i].rgbReserved = 0;

  ::SetDIBColorTable(m_pdc->GetSafeHdc(),
    0, iColors,
    pRGB);
SetPalette encapsulates a call to the GDI function::SetDIBColorTable. This function sets up an internal color table to map the colors in the DIB to equivalent RGB functions. This color table is not a Windows palette object.

SetPalette builds an array of RGBQUADS from the colors in the palette parameter. The array of RGBQUADS is passed to ::SetDIBColorTable.

That's all there is to CSimpleDIB and CreateDIBSection. You can use CDIBSurface or CSimpleDIB, or you can create your own class.

Pixel Format Choices

Rendering on DIBs instead of the window gives us a new choice: We can pick the number of bits per pixel we use for our DIB sections. However, we can't pick the number of bits per pixel for the display. How do you determine which format to use for your DIBs? Figure 7 shows the possible combinations.

<table>
<thead>
<tr>
<th>Format</th>
<th>8-bpp display</th>
<th>16-bpp display</th>
<th>24-bpp display</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDIBSurface</td>
<td>62</td>
<td>66</td>
<td>137</td>
</tr>
<tr>
<td>CSimpleDIB (8-bpp)</td>
<td>45</td>
<td>64</td>
<td>125</td>
</tr>
<tr>
<td>CSimpleDIB (16-bpp)</td>
<td>9909</td>
<td>891</td>
<td>843</td>
</tr>
<tr>
<td>CSimpleDIB (24-bpp)</td>
<td>140</td>
<td>670</td>
<td>44</td>
</tr>
<tr>
<td>CSimpleDIB (Identity)</td>
<td>46</td>
<td>60</td>
<td>143</td>
</tr>
</tbody>
</table>

Blt performance is affected by many factors, including the size of the DIB and the number of colors used in the DIB. Refer to Nigel's book or the WinG Programmer's Reference (in the Development Library, see Product Documentation, SDKs) for more information on blting performance.

A good compromise is to use the same number of bits per pixel for the DIB and the screen. This approach simplifies the code significantly, but there is a performance penalty on 16-bpp displays. For CMyStatusBar and CShapeMenu, I used 8-bpp DIBs except in 24-bpp display mode.

If you manipulate the memory in the DIB section directly, you might not want to use different pixel formats, because you'll have to write different code to manipulate the formats. If the application I'm writing modifies the pixel values directly, I prefer to support a single pixel format. I usually choose either 24-bpp or 8-bpp, because these formats are the easiest to work with.

A Few Choice Words About Palettes
Palettes have complicated this discussion. The general rule is that objects that have an 8-bpp format need a palette or some kind of color information. Objects that have 16-bpp or 24-bpp formats do not need palettes.

A DIB section is not a palettized device. DIB sections do not use palettes, but they do have a color table. The color table can be initialized in three ways:

- With colors in the \texttt{BITMAPINFO} structure during creation
- From the palette selected into the device context passed to \texttt{CreateDIBSection}
- With \texttt{::SetDIBColorTable}

Only 8-bpp DIBs have color tables—24- and 16-bpp DIBs do not. If an 8 bpp DIB does not have the color table initialized, the DIB section will be a black rectangle when it is blt to the screen.

Let's look at three cases:

- Blting an 8-bpp DIB section to an 8-bpp display
- Blting an 8-bpp DIB section to a 24-bpp display
- Blting a 24-bpp DIB section to an 8-bpp display

In the first case, we have an 8-bpp DIB section, which we will blt to an 8-bpp display. Because the DIB section is 8 bpp, the color table needs to be initialized. Regardless of whether we're using \texttt{CDIBSurface} or \texttt{CSimpleDIB}, we use the \texttt{::SetPalette} function, which calls \texttt{::SetDIBColorTable}. The display is also 8 bpp, so we need to select and realize a palette before we can display the DIB section, as shown in Figure 8.

![Figure 8. Palette use with 8-bpp DIB section and display](image)

The palettes should be the same if you created the DIB using \texttt{DIB_PAL_COLORS} in the \texttt{CreateDIBSection} call. If you used \texttt{DIB_RGB_COLORS} instead, GDI will map colors from the color table to the currently selected palette. This is slow and can result in really ugly displays, depending on the palettes. It's best to keep the number of palettes used by your application to a minimum. Using zero palettes is the easiest, but doesn't result in very colorful displays. Therefore, the best approach is to have one palette in your application.

The next case we'll examine is an 8-bpp DIB section and a 24-bpp display. Because the DIB section is 8 bpp, we need to initialize the color table in the DIB section. However, a 24-bpp display does not need a palette to display colors, so we do not need to select and realize a palette when displaying the 8-bpp DIB section on the screen, as illustrated in Figure 9.

![Figure 9. Palette use with 8-bpp DIB section and 24-bpp display](image)

The last case we'll look at is a 24-bpp DIB section and an 8-bpp display. We don't have to set the color table because a 24-bpp DIB does not need one. An 8-bpp display needs a palette to show more than 20 colors. Therefore, we must select and realize a palette before we can display our DIB section. This process is illustrated in Figure 10.
However, this method poses one problem: Where do you get the palette from? The \textit{CGL} class has been building all of our palettes for us, but \textit{CGL} does not create a palette when it creates a rendering context for a 24-bpp DIB. This means that it is the application's responsibility to create a palette. \textit{CGL} could be extended to generate a 3-3-2 palette on demand, just for this case. Because GDI will map the DIB section to the current palette when it is blting, the palette does not have to be a 3-3-2 palette, as discussed in the "\textit{OpenGL II: Windows Palettes in RGBA Mode}" article in the MSDN Library.

**Identity Palettes**

If you have read Nigel Thompson's book \textit{Animation Techniques for Win32}, you know that an identity palette is one of the keys to fast blt performance. An identity palette is a logical palette that is identical to the system palette. A logical palette is the palette you select into the DC. If you use a non-identity palette, GDI must get a color from the logical palette and then look for this color in the system palette. If you use an identity palette, GDI can simply use the logical palette indexes without any translation. In theory, bypassing the translation step may result in much better performance, as illustrated in Figures 11 and 12.

Although we can't use RGBA mode to get an identity palette, we can use color index mode. In color index mode, we specify each color by its palette index instead of using its RGB value. In RGBA mode, we can't pick the palette; in color index mode, we can. For more information on color index mode, see my "\textit{OpenGL IV: Color Index Mode}" article in the MSDN Library.

To make an identity palette using color index mode, you must construct the palette so that the first 10 and last 10 entries are available for the system colors. The other 236 colors are left up to you.
The `CSceneView::TimeIdentity` function creates an identity palette that you can use with a `CDIBSurface` object. I use Nigel’s `CDIBPal` class for its `SetSysPalColors` function, which creates an identity palette. The code in this section is taken from the `TimeIdentity` function.

The first step is to create a `CDIBSurface` object and render on it with OpenGL. This process is the same as we discussed previously in the section on `CDIBSurface`; the only difference is that we use `CSceneCI` instead of `CScene`. `CSceneCI` is a color index version of `CScene` and is discussed in the "OpenGL IV: Color Index Mode" article in the MSDN Library.

```cpp
CClientDC dc(this);
CDC* pDC = &dc;

CDIBSurface aDIBSurfaceIdent;
aDIBSurfaceIdent.Create(m_sizeBitmap.cx, m_sizeBitmap.cy, NULL);
CDC* pdcTemp = aDIBSurfaceIdent.GetDC();

CSceneCI aSceneCI; // Use color index mode.
aSceneCI.Create(pdcTemp);
CPalette* pPalTemp = aSceneCI.GetPalette();
if (pPalTemp) aDIBSurfaceIdent.SetPalette(pPalTemp);

pdcTemp = aSceneCI.GetDC();
CSceneCI aSceneCI; // Use color index mode.
aSceneCI.Create(pdcTemp);
CPalette* pPalTemp = aSceneCI.GetPalette();
if (pPalTemp) aDIBSurfaceIdent.SetPalette(pPalTemp);

pdcTemp = aSceneCI.GetDC();
CSceneCI aSceneCI; // Use color index mode.
aSceneCI.Create(pdcTemp);
CPalette* pPalTemp = aSceneCI.GetPalette();
if (pPalTemp) aDIBSurfaceIdent.SetPalette(pPalTemp);
```

Because I am going to use Nigel’s `CDIBPal` class, I must get the palette that `CSceneCI` creates into a form that `CDIBPal` can use.

`CDIBPal` requires a `LOGPALETTE` pointer at creation. Notice the use of the `PC_NOCOLLAPSE` flag below; this is to ensure that we get all 236 colors in our palette, even if they repeat.

```cpp
int iColors = 0;
pPalTemp->GetObject(sizeof(iColors), &iColors);
int iSysColors = pDC->GetDeviceCaps(NUMCOLORS);
LOGPALETTE *pPal = (LOGPALETTE *) malloc(sizeof(LOGPALETTE) + iColors * sizeof(PALETTEENTRY));
PALETTEENTRY* pe = pPal->palPalEntry;
pPal->palVersion = 0x300; // Windows 3.0
pPal->palNumEntries = (WORD) iColors; // table size
pPalTemp->GetPaletteEntries(0, iColors, pe);

for (i = 0; i < iSysColors/2; i++) {
    pe[i].peFlags = 0;
}
for (; i < iColors-iSysColors/2; i++) {
    pe[i].peFlags = PC_NOCOLLAPSE;
}
for (; i < iColors; i++) {
    pe[i].peFlags = 0;
}
```

We use the `LOGPALETTE` structure created in the code above to create a `CDIBPal` object, and we call `SetSysPalColors` to create the identity palette. We can now draw the object with an identity palette.

```cpp
CDIBPal aIdentPal;
BOOL bResult = aIdentPal.CreatePalette(pPal);
free (pPal);
aIdentPal.SetSysPalColors();
```

To look at the palette, you can use the following function:

```cpp
aDIBSurfaceIdent.Draw(pDC);
```

For debugging purposes, I wrote the following `IsIdentityPal` function in `CSceneView` to confirm that I really had a system palette:

```cpp
BOOL CSceneView::IsIdentityPal(CDC* pDC)
{
    int iColors = 0;
    PALETTEENTRY peCurrent[256];
    PALETTEENTRY peSystem[256];

    // Current logical palette
    CPalette* pCurrentPal = pDC->GetCurrentPalette();
    pCurrentPal->GetObject(sizeof(iColors), &iColors);
    peCurrent[0] = pCurrentPal->GetPaletteEntries(0, iColors, &peCurrent);

    // System Palette
    int iPals = pDC->GetDeviceCaps(SIZEPALETTE);
    ::GetSystemPaletteEntries(pDC->GetSafeHdc(),
```

```cpp
```
IsIdentityPalette gets the colors for the logical palette and the system palette, and compares them. Make sure that you have your logical palette selected and realized in the DC before calling IsIdentityPalette.

Nigel assures me that identity palettes really speed things up. My timing tests showed that identity palettes were faster than non-identity palettes, but not significantly, perhaps because the images that I was blitting in the test cases did not contain many colors. GDI may be caching the colors it looks up, resulting in increased performance for the non-identity palette cases.

Getting It Right

It's easy to make mistakes when rendering on a bitmap. This section provides a checklist of guidelines you can follow.

- When you use PFD_DRAW_TO_BITMAP, the DC passed to SetPixelFormat and wglCreateContext must be the same DC for all wglMakeCurrent calls. The rendering context is tied to the DC. When you use PFD_DRAW_TO_WINDOW, the rendering context is tied to the window, and any DC obtained from the window can be used in the "wiggle" functions.
- You must select the bitmap into the DC before calling SetPixelFormat.
- The DC must remain valid; you can't delete the DC and create a new one.
- The DIB section can't be resized. To resize a DIB, you must create a new DIB section and rendering context from scratch. If you want to paint the client area of a window with a DIB section that you are rendering with OpenGL, you must recreate the DIB section and rendering context. Another solution is to pick a DIB section as big as you will ever need, then use only the part that you need. However, this approach could waste a lot of memory.
- ChoosePixelFormat and SetPixelFormat require the correct number of bits per pixel. These functions will not examine a DIB section to determine the appropriate pixel format. You must pass the same number of bits per pixel to SetPixelFormat as you do to ::CreateDIBSection.
- Don't select a palette into the DC for the DIB section. A DIB section uses a color table. Selecting a palette into the DC of the DIB section results in incorrect output.
- If the DIB section is 8 bpp, initialize its color table. Objects that have 8-bpp pixel formats require color information. Use ::SetDIBColorTable, CSimpleDIB::SetPalette, or CDIBSurface::SetPalette.
- If the display is 8 bpp, select and realize a palette. To display an 8-bpp, 16-bpp, or 24-bpp DIB section on an 8-bpp device correctly, you must select and realize a palette first.
- Remember that CDIBSurface::BltBit and CDIBSurface::Draw select and realize the palette. This can lead to unexpected results if you don't keep it in mind.
- Remember that PFD_NEED_PALETTE is not set for displays that are not 8 bpp, even if PFD_DRAW_TO_BITMAP is set and the destination rendering surface is an 8-bpp DIB section.

Conclusion

Rendering OpenGL scenes on a DIB section gets OpenGL out of the client area and into menus and status bars. Once you have rendered an OpenGL scene on a DIB section, you can use any GDI function you desire to manipulate that DIB section. This includes blitting the DIB section to the screen, as well as drawing and printing text on the bitmap.

You can use bitmaps almost anywhere, so OpenGL-rendered bitmaps can easily appear in menus, status bars, button bars, and anywhere else you might want to use them.

Bibliography


Sources of Information on OpenGL

OpenGL VI: Rendering on DIBs with PFD_DRAW_TO_BITMAP The PFD_DRAW_TO_BITMAP pixel format descriptor flag allows OpenGLT applications to render to a Microsoft® Windows® deviceâ”“independent bitmap (DIB). The resulting DIB can be manipulated to the full extent using the commands in the Windows graphics device interface (GDI). This article explains how you can render OpenGL scenes on DIBs with PFD_DRAW_TO_BITMAP.Â OpenGL VII: Scratching the Surface of Texture Mapping This article explains how to apply bitmaps to OpenGLT surfaces to give them a realistic appearance. The bitmaps are known as textures and can resemble wood, marble, or any other interesting material or pattern. When setting pixel format descriptor dwFlags to PFD_DRAW_TO_BITMAP, where opengl renders and how to display that one on the screen? [This message has been edited by ramana (edited 12-15-2003).] 12-15-2003, 11:53 PM #2. See MSDN for the article "OpenGL VI: Rendering on DIBs with PFD_DRAW_TO_BITMAP" http://msdn.microsoft.com/library/de...l/msdn_gl6.asp. « Previous Thread | Next Thread ». Similar Threads. WGL_ARB_pbuffer vs. PFD_DRAW_TO_BITMAP. By gator in forum OpenGL: Advanced Coding. Replies: 4. Last Post: 05-17-2005, 01:15 PM. PFD_DRAW_TO_BITMAP Texture. By BingoWingo in forum OpenGL: Basic Coding. Replies: 0. Last Post: 11-19-2004, 06:57 AM. OpenGL VI: Rendering on DIBs with PFD_DRAW_TO_BITMAP. OpenGL VII: Scratching the Surface of Texture Mapping. OpenGL VII: wgliUseFontOutlines.Â Rendering Fast reflections with OpenGL. Achieving Quality PostScript output for OpenGL. Shadows, Lighting, Reflections, and Textures all at Once. Rendering a Magic Halo with OpenGL. A Simple OpenGL-based API for Texture Mapped Text. Virtualized OpenGL Light Sources for Efficient Multi-Light Source Lighting. Interactive Rendering of CSG Models (with OpenGL stencil). Implementing and Using BSP Trees: Fast 3-D sorting. Using paletted textures in OpenGL with MFC.