January 1997

MICROSOFT SYSTEMS JOURNAL

More Fun with MFC: DIBs, Palettes, Subclassing, and a Gamut of Reusable Goodies

I didn't just fill my viewer with more and more lines of open code, like the incredible expanding blob; rather, I encapsulated my solutions in reusable C++ classes-classes I can use over and over again, as is, in any app. In other words, I didn't just solve each problem specifically, I solved it generally.

Paul DiLascia

This article assumes you're familiar with C++, MFC, Win32.

Code for this article: MFCP1.exe (37KB)

Paul DiLascia is a freelance software consultant specializing in training and software development in C++ and Windows. He is the author of Windows ++: Writing Reusable Code in C++ *(Addison-Wesley, 1992).*

I started writing this article as a how-to piece about writing a Quick View file viewer in MFC. In the process of building my viewer, I had to confront a number of real-world Windows programming problems—drawing device-independent bitmaps, handling palette messages, dealing with fonts and so on. Nothing to do with writing file viewers per se, just everyday Windows® fare, the sort of stuff you'd expect to be trivial but that, because of the vagaries of Windows and MFC, are more likely to make you pull your hair out.

As I solved each problem, I didn't just fill my viewer with more and more lines of open code, like the incredible expanding blob; instead, I encapsulated my solutions in reusable C++ classes—classes I can use over and over again, as is, in any app. In other words, I didn't just solve each problem specifically, I solved it generally. By the time I was finished, I'd accumulated a handy grab-bag of reusable components useful for any application. So I decided to share the goodies with the rest of the world. Even more important, I hope to illustrate a particular style of software development, one that emphasizes creating reusable components as you build your app. Not that this is going to be some sort of academic OOP sermon—on the contrary, this will be a true-life case history, an exercise in real-world programming, where surprise lurks behind every function call and nothing ever goes the way you hope or want.

The Master Plan

When my client, the Maximum Leader of Acme Corporation, a multi-quadrillion-dollar conglomerate with

offices across the solar system, asked me to develop a file viewer for their XYZ documents, I said, "Sure thing, Max. No problem. When do you want it?"

"Three hours," he said, in a tone not to be trifled with.

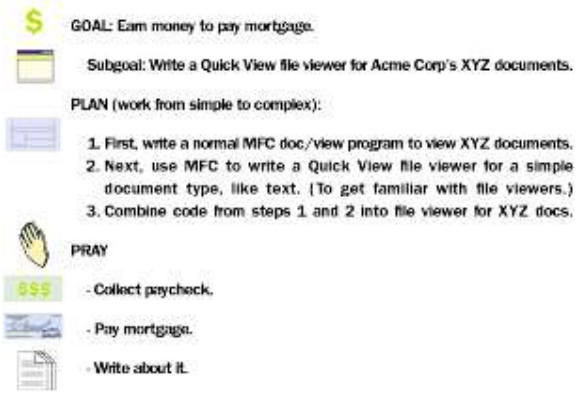

Figure 1 Master Plan of Attack

Gulp. How do you write a file viewer in three hours? Well, whenever you're confronted with a daunting project in life or programming, the best thing to do is break it up into manageable tasks. **Figure 1** shows my Master Plan of Attack. In this article, the first of two parts, I'll focus on step one of The Plan, writing the XYZ viewer. I'll show you all the goodies I developed along the way. In a later article, I'll show how I converted my XYZ viewer into a Quick View file viewer.

Since the format of Acme Corporation's XYZ documents is super top secret, known only to five people, I'll use DIBs (device-independent bitmaps) as my file format. I chose DIBs because the real XYZ documents contain them, because DIBs are fun (all those pretty pictures to look at), and because dealing with DIBs is a real pain in the tush.

The problem is that while DIBs are common on disk—almost any .BMP or .DIB file you're likely to run across uses the modern DIB format—internally, Windows still prefers the older DDBs (device-dependent bitmaps). To draw a DIB, you have to convert it to a DDB (HBITMAP) using CreateDIBitmap, or else draw it directly using the special functions SetDIBitsToDevice or StretchDIBits. MFC's CBitmap class does nothing to make any of this easier; it's just the C++ equivalent of HBITMAP.

As my main project for this article, I'll show you how to write a program called DIBVIEW that displays DIBs. Since I'm a firm believer that the easiest way to write complex programs is to first write simple ones and then extend them, I wrote DIBVIEW in a series of steps (see **Figure 2**). Each step builds on the previous one, just like the infamous SCRIBBLE tutorial. That way it's easier for me to write the code and for you to understand it. Unfortunately, I can't show you all the steps because, if I did, this issue of *MSJ* would be thicker than an issue of Modern Bride. Only step three is printed here; next month I'll show you step four, the final one.

CDib and The Baby Bitmap Viewer

OK, let's start writing DIBVIEW. Not a Quick View file viewer, but just an ordinary old standalone app. There's already a DIB viewer in the MFC SAMPLES directory, but using DIBLOOK would be cheating. Besides, that code is so old and grungy even Rip Van Winkle wouldn't recognize it, not to mention the fact that it doesn't even display hi-color bitmaps! All the Redmondtonians did for DIBLOOK was copy the code from an even older C SDK sample—called coincidentally DIBVIEW—and make it look marginally like C++. Our DIBVIEW uses all the

latest APIs and it's object-oriented to the core. Plus, you can use the basic DIB class in any app.

Since the ultimate aim is to build a Quick View file viewer, and file viewers view only one doc at a time, I implemented DIBVIEW as an SDI app. Just select "Single Document" in AppWizard and press the GO button to get an instant vanilla app. The only thing left to do is fill in the blanks. Aside from all the fancy features, this basically means: write some code to load the bitmap and display it. So, where to begin?

When in doubt, define a class. Preferably with a suggestive name like CDib, and some equally suggestive functions like Load and Draw.

```
// Device Independent Bitmap
class CDib : public CObject {
public:
   BOOL Load(LPCTSTR szPathName);
   BOOL Draw(CDC* pDC);
};
```

CDib is the first of several goodies you'll meet in this article. I built it for DIBVIEW, but you can use it in any app that uses DIBs. I'll show you two different implementations of CDib, one quick-and-dirty and the other more thorough. That's how I did it in real life and I want to emphasize how, if you encapsulate your code properly, you can change the implementation of a class without affecting the rest of the app much, if at all.

For my first implementation, I was in a hurry (only three hours to write the whole thing). I didn't want to mess with CreateDIBitmap or palettes or whatever. I just wanted my DIB on the screen. This is where my experience with Video for Windows (VFW) came in handy. VFW has a really great API called DrawDib made just for drawing DIBs. VFW uses DrawDib to draw the frames in a video sequence, but you can use it to draw individual DIBs too. DrawDib is fast, and even does decompression. Most important, it's easy to use. There's a single function, DrawDibDraw, that draws the DIB. But before you can call it, you have to load the DIB into memory.

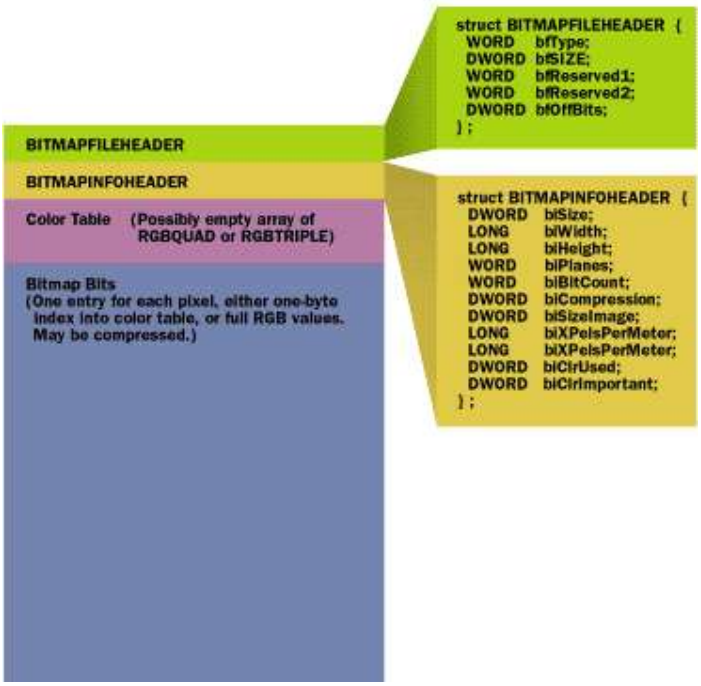

Figure 3 Anatomy of a DIB

To load a DIB, you have to know its format, which is well documented in the Windows manuals. A bitmap comprises three parts: a BITMAPINFOHEADER, a color table or "palette," and the image bits themselves (see **Figure 3**). For DIBs on disk, there's also a BITMAPFILEHEADER to precede all this. If the bitmap has a color table, the image bits are single-byte indexes into the color table. Bitmaps with more than 256 colors don't have a color table, they just use RGB values for each pixel. They take more space, but the color is better. You can find out if a bitmap has a color table by looking at information in the BITMAPINFOHEADER (more on this later.)

Figure 4 shows my first cut at CDib, including the Load function. Actually, there are two Load functions: CDib::Load(LPCTSTR) opens a CFile on the given path name, then calls CDib::Load(CFile&) to do the work. I broke it up this way in case I ever wanted to load a DIB directly from a CFile. Either way, the first step in loading is to read the BITMAPFILEHEADER.

```
BITMAPFILEHEADER hdr;
DWORD len = file.Read(&hdr, sizeof(hdr));
if ((len!=sizeof(hdr)) ||
      (hdr.bfType!=BITMAP_TYPE)) {
   TRACE0("***CDib: bad BITMAPFILEHEADER\n");
   return FALSE;
}
```

If the information in the header doesn't pass a few rudimentary checks, CDib::Load gives up and returns FALSE. In particular, the first two bytes (hdr.bfType) must have the magic number BITMAP_TYPE, which is "BM" in ASCII.

Assuming the header is OK, the next step is easy: just read the rest of the file into a buffer.

```
len = file.GetLength() - len;
m_pbmih = (BITMAPINFOHEADER*)new char[len];
file.Read(m_pbmih, len);
```

CDib::m_pbmih points to the BITMAPINFOHEADER and everything else following it (color table and bitmap bits). Once the bitmap is loaded, drawing it is easy with DrawDib.

```
BOOL CDib::Draw(CDC& dc,
   const CRect* rcDst, const CRect* rcSrc)
{
   if (!m_hdd)
      VERIFY(m_hdd = DrawDibOpen());
   return DrawDibDraw(m_hdd, dc,
                      ...,
                      GetBits(),
                      ...);
}
```

CDib::Draw calls DrawDibOpen to get a handle to a DrawDib context, then calls DrawDibDraw with this handle, the HDC, and a bunch of arguments: rectangle coordinates, pointer to the BITMAPINFOHEADER, pointer to the bits, and some flags. It's not every day you get to use a function that takes thirteen arguments! DrawDib does whatever decompression and bit-stretching are required to draw your DIB on the DC. The only hard part is getting the pointer to the actual bitmap bits in memory.

```
LPBYTE CDib::GetBits()
{
  return (LPBYTE)m_pbmih + // start of bitmap +
    m_pbmih->biSize +      // size of header +
    GetNumPaletteColors()  // (num colors *
```

```
      *sizeof(RGBQUAD);    // size each entry)
}
```

The bits come after the BITMAPINFOHEADER and the color table, so getting them requires computing the size of the color table. Once you have that, just multiply by the size of each entry, which is an RGBQUAD. In older bitmaps (pre-Windows 3.0), the table is an array of RGBTRIPLE instead of RGBQUAD, but supporting old bitmaps was not a requirement for me, so I made Load reject them. (My second implementation supports old bitmaps too—stay tuned.) To compute the number of colors, I wrote yet another function.

```
UINT CDib::GetNumPaletteColors()
{
   UINT nColors=m_pbmih->biClrUsed;
   if (nColors==0 && m_pbmih->biBitCount<=8)
      nColors = 1<<m_pbmih->biBitCount;
   return nColors;
}
```

The biClrUsed field in the BITMAPINFOHEADER tells how many colors are in the color table—unless it's zero, in which case you have to compute the number of colors by looking at how many bits are used per pixel (biBitCount) and raising 2 to that power. For example, 4-bit color implies 16 total colors; 8-bit color implies 256 total colors. For more than 8-bit color, there's no color table at all because the image bits contain actual RGB values. All this stuff is boring and well-documented, so I won't dwell on it. The important thing is to observe how I started with a simple class, CDib, with two functions, Load and Draw, and expanded it as necessary to implement them.

Since most of the time I plan to draw my DIBs without stretching, but sometimes I'll want to, I added source and destination rectangles as arguments to my Draw function, with default NULL values, which tells Draw to use the default. CDib::GetSize returns the width and height as a CSize object.

With CDib implemented, the rest of DIBVIEW is pretty brainless. I added a CDib as a data member in my document class, CDIBDoc, then overloaded CDIBDoc::OnOpenDocument to call CDib::Load.

```
BOOL CDIBDoc::OnOpenDocument(LPCTSTR lpszPathName)
{  DeleteContents();
   return m_dib.Load(lpszPath);
}
```

To draw the bitmap, I call CDib::Draw from my view's OnDraw function.

```
void CDIBView::OnDraw(CDC* pDC)
{
   CDib* pDIB = GetDIB();
   if (pDIB)
      pDIB->Draw(*pDC);
}
```

I derived my view class from CScrollView to support scrolling, so there's also the usual business in OnInitialUpdate to set the scroll sizes to the width and height of the DIB, and I wrote CDIBDoc::DeleteContents to delete my DIB.

```
void CDib::DeleteObject()
{
  if (m_hdd) {
    DrawDibClose(m_hdd);
    m_hdd = NULL;
  }
  if (m_pbmih) {
    delete [] (char*)m_pbmih;
```

```
      m_pbmih = NULL;
    }
  }
```

Figure 5 DIBVIEW

Figure 5 shows DIBVIEW in action. Not bad for a few hours work, huh?

CDib Redux

As soon as I got DIBVIEW up and running—not a nanosecond later—I raced over to Acme World Headquarters to present it to His Supremeness the Maximum Leader. He was not especially impressed, but after much cajoling I managed to persuade him to give me another 48 hours to write the viewer.

Back in my programming cave, I decided to ignore all time pressures and reimplement CDib. Why? Because there was something about CDib that bothered me, something I knew would cause trouble down the road. Since a device-independent bitmap "is a" bitmap, CDib should be derived from CBitmap, not CObject.

```
CObject     // base for all objects
 CBitmap    // "ordinary" bitmap
  CDib      // DIB
```

This isn't just a philosophical consideration; it has real implications. I should be able to use a CDib anywhere I can use a CBitmap. If SomeFunction takes a pointer to a CBitmap as an argument, I should be able to pass a CDib as well. Why not? To do this, CDib must be derived from CBitmap. So I froze my baby bitmap viewer as step one and began step two by altering the derivation of CDib.

```
class CDib : public CBitmap {
   .
   .
   .
};
```

What this means in practice is that I must somehow convert my DIB into an HBITMAP, since that's all CBitmap really is: a wrapper for HBITMAP.

One way to do it is to call CreateDIBitmap, but Win32® has a handy new function that makes it almost trivial to load a DIB and create an HBITMAP handle to it.

```
// revised load function
BOOL CDib::Load(LPCTSTR lpszPath)
{
```

```
  return Attach(::LoadImage(NULL, lpszPath,IMAGE_BITMAP,
                            0, 0, LR_LOADFROMFILE |
                            LR_CREATEDIBSECTION | LR_DEFAULTSIZE));
}
```

LoadImage is a multipurpose Win32 API function that loads icons, cursors, and bitmaps. There are all sorts of different ways you can use LoadImage (see the Win32 documentation); for CDib, I use IMAGE_BITMAP to tell LoadImage to load a bitmap, not an icon or cursor, and LR_LOADFROMFILE to load it from a file instead of a resource. The width and height are zero because I'm using LR_DEFAULTSIZE to load the bitmap in its normal size, and LR_CREATEDIBSECTION tells LoadImage to load the bitmap as a DIB. Without LR_CREATEDIBSECTION, LoadImage would map my bitmap's colors to match the current display device. That can be handy sometimes, but definitely not for CDib.

Assuming it can read the file, LoadImage returns a handle to the bitmap (HBITMAP). What could be easier? CDib::Load passes this handle to Attach (actually CGdiObject::Attach) the same way you can call CWnd::Attach to manually attach an HWND to a CWnd object.

That's it—that's all there is! Now my CDib is a bonafide CBitmap, all hooked up with a real HBITMAP and everything. One major benefit of using LoadImage instead of loading the DIB myself is that now I don't have to worry about those pre-Windows 3.0 bitmaps. LoadImage can handle it. Not to mention that my Load function has gone from about thirty lines of code to about one. In fact, you're probably wondering, "Why didn't you use LoadImage in the first place, you dimwit?" That's only because you haven't seen what's required to draw the DIB yet!

You'd think that after all these years, with bitmaps as ubiquitous as they are in Windows, there'd be a function somewhere to just draw a lousy bitmap on the screen. But no, you have to go through minor conniptions to do it. First, you create a memory device context compatible with the display, then you select the bitmap into the memory device context, then you call BitBlt to copy the bits to the target device context. In C++, it looks like this:

```
BOOL DrawBitmap(CDC& dc, CBitmap* pbm)
{
   CDC mdc; // memory DC
   mdc.CreateCompatibleDC(&dc);
   CBitmap* pOld = mdc.SelectObject(pbm);
   BOOL bRet = dc.BitBlt(...);
   mdc.SelectObject(pOld);
   return bRet;
}
```

I left out the args to BitBlt because there's a ton of them, mostly coordinates. If stretching is required, you use StretchBlt instead of BitBlt. It's not back-breaking, but it is several lines of code. Even that isn't enough; DrawBitmap as I've shown it works for ordinary CBitmap objects, but fails for DIBs. Why? Because to draw a DIB, you must first select, then realize, its palette. This is what makes a DIB a DIB—that is, device independent.

Figure 6 Palette-free DIBVIEW

A DIB specifies true RGB values, either directly or through a palette. If you implemented your draw function as above, **Figure 6** is what you'd get. Not the sort of thing to show Maximum Leader. With no palette, Windows draws the bitmap using the 20 standard system colors—red, green, cyan, magenta, and so on. To get the DIB's true colors, you must select its palette.

```
BOOL CDib::Draw(CDC& dc)
{
  CPalette* pPal = ?
  CPalette* pOldPal =
    dc.SelectPalette(pPal,FALSE);
  dc.RealizePalette();
  BOOL bRet = DrawBitmap(dc, this); // as before
  dc.SelectPalette(pOldPal, TRUE);
  return bRet;
}
```

First you select, then "realize," the palette before drawing the bitmap using the DrawBitmap function I just showed you. But this still isn't the end of the story because, as you may have noticed, I didn't tell you where to get the palette. In fact, this turns out to be quite a mystery. Nowhere in the Windows API will you find a function to get a bitmap's palette. It's true. You have to create the palette manually by copying the DIB's entire color table into a special LOGPALETTE (logical palette) structure, then calling CreatePalette with this structure. You even have to take into account whether the bitmap uses the older RGBTRIPLE (three bytes) or newer RGBQUAD (four bytes) to hold the color values.

Win32 has a new function, GetDIBColorTable, that gives you the colors, always in RGBQUAD format. You still have to copy them into your LOGPALETTE. There's just one catch. Before you can call GetDIBColorTable, you must have the bitmap selected in a device context somewhere, so once again you have to create a memory DC. In summary, here's what you have to do to get the DIB's palette.

- Create memory DC
- Select bitmap into it
- Allocate array of RGBQUAD
- Call GetDIBColorTable to fill array
- Allocate LOGPALETTE array
- Copy each RGBQUAD entry to LOGPALETTE
- Call CreatePalette with LOGPALETTE
- Delete RGBQUAD array

- Delete LOGPALETTE struct

I implemented all this in a new function, CDib::CreatePalette, which creates the palette in a new data member, m_pal. I also overrode the Attach function to create the palette whenever one of my Load functions attaches a new HBITMAP to the CDib object. Now that I finally have the palette, I can select it and draw the bitmap. Whew!

There's just one more little trick I didn't tell you about. Doing the RGBQUAD/LOGPALETTE schtick works fine for bitmaps that have color tables (palettes)—but what about hi-color bitmaps with 24 or 32 color bits? Those DIBs don't have a color table. Instead, each pixel has a full RGB value, GetDIBColorTable returns zero as the number of entries in the color table, and there's no palette. So what're you going to do? You still need a palette—you saw what happens without one in **Figure 6**. You could analyze the bitmap and select the best 256 colors but, sheesh, that's ridiculous—and I've only got 40 hours left! Do the folks in Redmond really expect me to write a zillion lines of code just to compute a palette?

Figure 7 256 colors, no palette selected

No, you only have to write one line. You can create something called a halftone palette, which is essentially just a palette with a broad range of colors in it that looks good for most hi-color DIBs. So if CDib::CreatePalette encounters a DIB with no palette, it creates a halftone palette. **Figure 7** shows an example of a 24-bit color bitmap displayed using the system palette (yech!); **Figure 8** shows the same image displayed using a halftone palette. Thanks to Jeff Prosise for this wicked trick.

Figure 8 256 colors, halftone palette

I've written a lot of code, but now CDib is general enough to be useful. It has Load and Draw functions that handle all kinds of DIBs, and since it's derived from CBitmap (and implemented as one), you can use a CDib object anywhere you could use a CBitmap. As a fringe benefit, I exposed ::DrawBitmap, the function that draws an ordinary CBitmap, as a global extern function so you can use it to draw run-of-the-mill CBitmap bitmaps. In an ideal world, DrawBitmap would be a member function, CBitmap::Draw, not a global extern

function—but alas, the world is far from ideal and I'm not about to modify the MFC source code.

Best of Both Worlds

As soon as I implemented my new, improved CDib class, I thought it would be interesting to compare it with my old DrawDib implementation. When I opened ordinary 256-color bitmaps, there was no noticeable difference, except I think the improved version is a little faster (I'm not sure I can really tell; it may be wishful thinking). When I opened 24-bit images, however, there was an obvious difference. **Figure 8** shows a 24-bit color image displayed with my new improved CDib class; **Figure 9** shows the same image displayed with the old version that uses DrawDib (both with my display set at 256 color mode); and **Figure 10** shows the image displayed with true 32-bit color turned on.

Figure 9 256 colors, halftone palette, dithering

Figure 10 32-bit true color

One thing my mama taught me is if you show a turned on client a program, and then you show him version two, the second version should be better, not worse. What's going on? Does DrawDib know something I don't?

Yup. DrawDib does dithering. That's a graphics technique of mixing the pixels to make them look better, so you don't get the abrupt color transitions in **Figure 8**. Windows 95 uses dithering to turn every other pixel grey when you shut down. You can still see the desktop, but it looks darkened. Color dithering is similar, but you dither the pixels with their neighbors instead of a fixed color like grey. It's a lot of work, but DrawDib does it. So the problem in a nutshell is that while my new CDib class is better from a programming perspective (because it derives from CBitmap), the old version looks better with hi-color images (because it uses DrawDib, which does dithering). Isn't there some way I can have my cake and eat it too?

Yes. In the final version of CDib (files Dib.h and Dib.cpp in **Figure 11**), I added a bunch of extra arguments to CDib:: Draw. (For example, you can pass a palette to use instead of the one that comes with the bitmap, in case you want to do weird palette effects, and you can tell CDib::Draw whether to draw in the

foreground or background.) One of the new arguments is bUseDrawDib. If you specify TRUE (the default), CDib draws the old way, using DrawDib; if not, it uses BitBlt or StretchBlt. Good thing I saved the old code! I had to write a few lines to get the BITMAPINFOHEADER and color table for DrawDibDraw since the CBitmap implementation no longer has the DIB in memory.

```
DIBSECTION ds;
GetObject(sizeof(ds), &ds);
char buf[sizeof(BITMAPINFOHEADER) + MAXPALCOLORS*sizeof(RGBQUAD)];
BITMAPINFOHEADER& bmih = *(BITMAPINFOHEADER*)buf;
RGBQUAD* colors = (RGBQUAD*)(&bmih+1);
memcpy(&bmih, &ds.dsBmih, sizeof(bmih));
GetColorTable(colors, MAXPALCOLORS);
```

The first time I wrote this, I allocated the combined BITMAPINFOHEADER/color table from the heap instead of the stack, and only the first time it was needed, because I thought it would slow things down to copy on every Draw. But when I tried it as above there was no perceptible performance hit, which is not that surprising when you think of how many bits get copied all over the place to draw a DIB. What's an extra kilobyte among friends? So I kept the stack implementation to save memory.

I made bUseDrawDib an argument instead of always using DrawDib because some images may look better without dithering and because dithering takes longer. Not noticeable on the screen, but definitely so for printing (more on this in Part II). I even propagated the bUseDrawDib flag up to the user interface as a new unrequested DIBVIEW feature: a View Dither command to let the user choose whether to dither or not.

Palette Problems in Paradise

By now things were really rolling. The improved CDib took a couple of days, leaving me with only one hour to implement all the other features and convert the thing into a file viewer COM object. Sensing that was unlikely, I showed Mr. Maximum Leader my latest DIBVIEW with the dithering feature and told him it was a super-sophisticated graphics technique that would catapult Acme Corp far beyond the competition. He was skeptical at first, but agreed reluctantly to give me another couple of days to write the viewer.

Figure 12 Two DIBVIEWs; two DIBs

Back in my cave, with time pressure removed, I took a few minutes to smell the roses. That is, gaze at some pretty bitmaps with my new viewer. Well, wouldn't you know it, just when I was feeling proud of myself for being so clever and productive, I discovered a slight problem. Well, er, OK, so it wasn't a slight problem, it was a major bug. **Figure 12** shows what happened

when I ran two instances of DIBVIEW with two different DIBs. Gak! Good thing Mr. Maximum Leader didn't see that! What's going on? Why the psychedelic snowsurfer?

The bug has to do with palettes. To produce it, I set my desktop display properties to 8-bit color mode, which means my system has a maximum of 256 colors it can display at any given time. Which particular 256 colors those are, out of the total possible of 224 = 16M colors that can be expressed with 8-bit RGB values, is called the current physical palette or system palette. (Actually, you only get 236 colors to play with because 20 colors are fixed for the basic colors like black and red —see **Figure 13**.) The psychedelic look comes from the fact that my snowsurfer has one idea about which 256 colors are in the system palette, while the Les Paul guitar has another. Whichever window has the focus wins. As for the loser, well, its color indexes end up pointing at who knows which colors in the system palette. The surfer window might think, for instance, that the 134th color in its palette is RGB(12,34,104) when in fact the guitar window has changed it to something completely different like RGB(205,0,19). When every color in surf.dib gets randomly mapped this way, the result is the brain salad palette in **Figure 12**.

Now, everyone knows the ideal way to solve this problem. Just examine the system color resolution when your program starts up and, if it's anything less than 24 bits, display a message box that says "CHEAPSKATE!" I mean, it's 1997. Anyone who can't afford a display card with enough memory to handle hi-res color doesn't deserve to look at DIBs anyway.

Well, OK, so that's not an option. But you should understand that this whole business with palettes—in fact, the very existence of palettes—is only a hack invented long ago to overcome feeble display cards. When you run in 24-bit or higher color modes, the palette problem disappears (as does the need for halftone palettes). Every window can have all the colors its little heart desires. Some day after the millennium, when the last 8-bit color card gets tossed into the nearest cosmic incinerator, the Friendly Redmondtonians will quietly pull the plug on palettes by turning all the palette functions in Windows 2000 into no-ops—just like they did with all the memory locking functions when 16-bit pointers bit the dust. Until then, you have to learn to realize your palettes in the background.

Since this is an article about building reusable goodies, not palette internals, I won't go into all the nitty-gritties of logical and physical palettes, but concentrate on fixing the problem. (For more info, see "The Palette Manager: How and Why" on MSDN.) The gist of it is that while many windows can each use different logical palettes, there's only one physical palette for the whole system at any given time.

You may have noticed a little Boolean argument I glossed over when I showed you how to select the palette.

```
dc.SelectPalette(pMyPal, FALSE);
dc.RealizePalette();
```

That FALSE tells Windows not to realize the palette in the background; in other words, realize it in the foreground. When you realize a palette in the foreground, Windows changes the system palette so it contains the colors in your logical palette. Of course, that throws all the other windows out of whack because now their logical palettes are no longer realized. To avoid the Timothy Leary look, inactive windows must

realize their palettes in the background, which they do by calling RealizePalette with TRUE instead of FALSE. Now Windows does something totally different; instead of altering the system palette, Windows remaps the logical palette to use colors already in the system palette, in the best possible way. For example, if a pixel in surf.dib wants to be a particular shade of blue that doesn't exist in the system palette, Windows will find the closest shade of blue it can, and use that instead. It's not perfect, but it's a lot better than some random hue in the spectrum. **Figures 14** and **15** show what two DIBs each look like with their palettes realized in the foreground and background.

Figure 14 Virgil's palette realized in background

Figure 15 Surfer's palette realized in background

So the basic idea is to realize your palette in the foreground when your window has focus, and in the background when it doesn't. What could be simpler? Alas, as with all things in Windows, the details may make you crave some Advil.

To give apps a chance to realize their palettes at the right time, Windows broadcasts two messages to all top-level windows. Windows sends WM_QUERYNEWPALETTE to your main frame window as it's about to activate your app. This is your big chance in life to realize your foreground palette. When you do (by calling Select/RealizePalette), Windows is likely to change some of the colors in the system palette. If so, Windows immediately broadcasts the other message, WM_PALETTECHANGED, to all top-level windows. This gives inactive apps a chance to rerealize their palettes in the background and repaint themselves using the new palette—thus avoiding the Leary look. Most apps ignore WM_QUERYNEWPALETTE and WM_PALETTECHANGED since most apps don't use palettes.

Armed with this new information, I set out to fix the palette bug. Once again, I froze DIBVIEW, called it step two and began work on step three. That way if I broke anything (as I am prone to do), I could always go back. The first thing I did was add a couple of message

handlers, CMainFrame:: OnQueryNewPalette and OnPaletteChanged. To get started, I copied some lines from DIBLOOK (OK, I admit it) and hacked them to fit my program.

Debugging Detour

Naturally, the first time I ran my palette code, it failed. Sigh. To find out why, I needed a picture of what messages my app was getting when. Spy++ is too cumbersome; all I really needed was a few diagnostic messages. So I sprinkled my code with TRACE macros and ran it again. As the barrage of messages flooded my screen, I realized this was no good; what I really needed to unravel all the palette pandemonium was some way to indent the messages to show the call stack. This may seem obsessive, but proper formatting goes a long way toward making text more readable. Besides, I'm an obsessive kind of guy.

There are several possible approaches to indenting debug output, but the quickest I could think of was to modify the MFC function ::AfxTrace, which is what all the TRACE macros call. The MFC developers were kind enough to put AfxTrace in its own file, dumpout.cpp, with only one other function, AfxDump (whose implementation is trivial), so all I had to do was copy the functions into my own debug module, debug.cpp (shown in **Figure 11**), and modify AfxTrace. My modified version formats the output string as before, but before it ships the string to the display, it scans it for newlines (\n) and inserts n spaces after each newline, where n is the current indent level, stored as a static global in a little class I invented.

```
class CTraceFn : public CFile {
private:
  static int  nIndent; // current indent
  friend void AfxTrace(LPCTSTR lpszFormat, ...);
public:
  CTraceFn()  { nIndent++; }
  ~CTraceFn() { nIndent--; }
};
```

I declared AfxTrace a friend of CTraceFn so AfxTrace can look at nIndent, which is private. The CTraceFn constructor increments the indent level; the destructor decrements it. The idea is that you instantiate one of these little doodads at the top of each function you want to TRACE.

To make using CTraceFn really easy and look more familiar, I defined a macro:

```
#define TRACEFN CTraceFn __fooble; TRACE
```

Now whenever I want to trace the action in a group of functions, all I have to do is add a TRACEFN statement at the top of each one.

```
void SomeFunction()
{
   TRACEFN("SomeFunction\n");
   .
   .
   .
}
```

The macro expands to:

```
CTraceFn __fooble; TRACE("SomeFunction\n");
```

That is, TRACEFN instantiates an instance of CTraceFn as an automatic variable __fooble, then expands whatever follows using TRACE. The constructor for __fooble bumps the indent level so the string "SomeFunction" appears indented in the debug output, as do all subsequent TRACE messages, until control flows out of SomeFunction and C++ calls the destructor for __fooble, which unbumps the indent level. If SomeFunction calls SomeOtherFunction that also has a TRACEFN statement, the output is indented twice. Pretty neat.

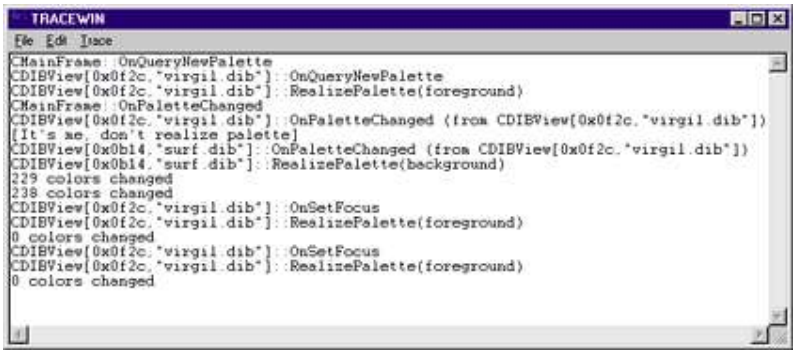

Figure 16 TRACE output without indenting

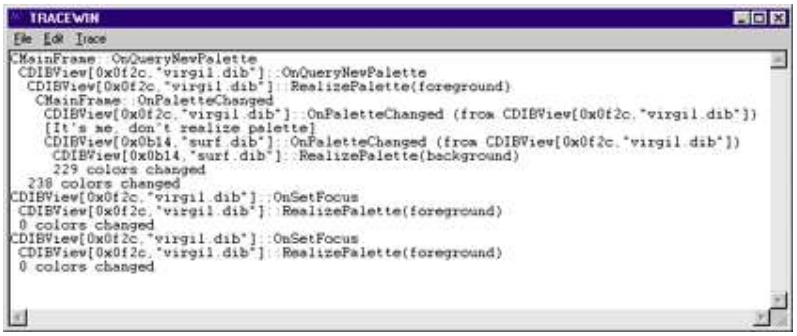

Figure 17 TRACE output with indenting

TRACEFN is simple, easy, and it makes diagnostic messages much more readable, as the before and after shots in **Figures 16** and **17** show. There's just one catch: You can't use TRACEFN in one-line if statements because it generates two C statements.

```
if (bletch)
   TRACEFN(...);
```

expands to

```
if (bletch)
   __fooble; TRACE(...);
```

which is equivalent to

```
if (bletch)
   __fooble;
TRACE(...);
```

which is probably not what you want. The solution is to always put TRACEFN in curly brackets.

```
if (bletch) {
   TRACEFN(...);
}
```

While I'm on the subject of debugging, let me tell you about another debugging goodie I implemented: sDbgName. This function returns a friendly name for a CWnd object as a CString. If you write

```
TRACE("Window is %s\n", (LPCTSTR)sDbgName(pMyWnd));
```

you'll get something like

```
Window is CDIBView[0x0cc4,"surf.dib"]
```

sDbgName generates a string in the form classname [hwnd, "title"]. sDbgName gets the class name from the object's runtime class; to get the title, it calls GetWindowText. If the window has no title (views typically don't), sDbgName looks at the parent and all other parents until it finds one with a title—that is, one for which GetWindowText returns a nonempty string. To avoid casting sDbgName to LPCTSTR every time I use it, I wrote another macro:

```
#define DbgName(x) (LPCTSTR)sDbgName(x)
```

Normally, you can pass CString anywhere const char* is expected without casting since the compiler automatically calls CString's conversion operator for you. But printf, AfxTrace, and CString::Format all have the "..." variable-number-of-arguments declaration, for which the compiler does no conversions since it has no type information. To pass a CString to printf, TRACE, or CString::Format, you must explicitly cast it to LPCTSTR or you'll get garbage output.

Since I sometimes find it useful to display the names of WM_XXX messages, I also wrote an overloaded sDbgName(UINT) that does just that. So instead of wondering what 0x0007 means in my diagnostic stream, I'll see WM_SETFOCUS. sDbgName(UINT) knows about private MFC messages like WM_IDLEUPDATECMDUI and WM_INITIALUPDATE, and displays them with an asterisk as in *WM_INITIALUPDATE. It displays messages beyond WM_USER in the form WM_USER+n. The HOOK program described later uses sDbgName(UINT) to spy on itself. Note that by using the same overloaded name (sDbgName) instead of using different names like DbgNameWnd and DbgNameMsg, my DbgName macro still works.

To MDI or Not to MDI?

Before getting back to palettes, I have to tell you about another little problem I ran into in step three. I mentioned earlier that I implemented DIBVIEW as an SDI app because my ultimate aim was to write a file viewer and file viewers are SDI apps. But I figured it would be nice to have a standalone viewer that supported MDI. I know from experience that palette handling is slightly more complex in an MDI app and I wanted to solve the palette problem generally, in a way that would work for MDI as well as SDI apps. So the next thing I did in step three was expand DIBVIEW to a MDI app. In order to preserve the SDI version and still have a single code base, I used #ifdef statements with a #define symbol, _MDI, to control the compilation. This worked well since there are surprisingly few places where differences between SDI and MDI apps show up in the source.

```
#ifdef _MDI
#define CBaseFrameWnd CMDIFrameWnd
#else
#define CBaseFrameWnd CFrameWnd
#endif
```

```
class CMainFrame : public CBaseFrameWnd {
.
.
.
}
```

Also, the document templates are different and the startup code in CApp::InitInstance is slightly different too. No big deal, #ifdef took care of it.

Things were a little more troublesome when I got to the resource file. I tried to use the same _MDI symbol with conditional #ifdefs to build different menus, strings, and other resources for IDR_MAINFRAME.

```
IDR_MAINFRAMEMDI MENU PRELOAD DISCARDABLE
BEGIN
#ifdef _MDI
.
.
.
#else
.
.
.
#endif
END
```

This worked fine as far as the resource compiler was concerned, but when I edited my resources with App Studio, Visual C++® stripped all my #ifdefs—without even telling me! Whichever block in the #ifdef clause was not active when I opened the RC file got totally trashed! App Studio just threw it away as if it never existed! Boy, that's what I call unfriendly. So consider this a warning: be careful using #ifdef in resource files edited with Developer/App Studio! Lucky for me, I had backups (I don't trust anybody).

To work around Visual C++, I put all the versions of all the resources in my resource file using two different IDs, IDR_MAINFRAMESDI and IDR_MAINFRAMEMDI.

```
IDR_MAINFRAMEMDI MENU PRELOAD DISCARDABLE
.
.
.
END
IDR_MAINFRAMESDI MENU PRELOAD DISCARDABLE
.
.
.
END
```

I did the same for all the other resources so that my resource file has no conditional compile flags. Then, in DibView.h, I select which one to use as follows:

```
#ifdef _MDI
#define IDR_MAINFRAME IDR_MAINFRAMEMDI
#else
#define IDR_MAINFRAME IDR_MAINFRAMESDI
#endif
```

The only drawback is this wastes space since each version of DIBVIEW—SDI and MDI—contains all the other version's resources as well as its own.

Back to Regularly Scheduled Palette Programming

Armed with debugging tools and a dual-GUI app, let's return to the palette problem. While you were reading about TRACEFN, DbgName, and MDI, I was busy hacking DIBVIEW. I added OnQueryNewPalette and OnPaletteChanged handlers with TRACEFN macros to show what's going on (see MainFrm.cpp in **Figure 11**). I'll step you through the code in two different scenarios. If you consider yourself a palette guru, feel free to skip right along to the next section.

In the first scenario, DIBVIEW is running in an inactive state with two documents open when I click the mouse to activate it. The TRACE output in **Figure 18** records the action, which begins when I click on the DIBVIEW main window. Windows sends my app WM_QUERYNEWPALETTE, which MFC routes to my CMainFrame handler.

```
BOOL CMainFrame::OnQueryNewPalette()
{
  TRACEFN("CMainFrame::OnQueryNewPalette\n");
  CView* pView =
    GetActiveFrame()->GetActiveView();
  if (pView)
    pView->SendMessage(WM_QUERYNEWPALETTE);
  return FALSE;
}
```

Since the main frame itself doesn't draw anything—only the views do the drawing—CMainFrame delegates the message to the active view. Windows sends WM_QUERYNEWPALETTE to top-level windows only, but there's no reason you can't pass it along to your children. Note that CMDIFrameWnd::GetActiveFrame returns the active MDI child app, whereas CFrameWnd::GetActiveFrame returns "this"—that is, the CFrameWnd itself—so the above code works whether CMainFrame is derived from CMDIFrameWnd or CFrameWnd, whether my app is compiled for SDI or MDI. Either way, CMainFrame sends WM_QUERYNEWPALETTE to the active view. In this case, it's surf.dib.

Here's my CDIBView handler.

```
BOOL CDIBView::OnQueryNewPalette()
{
   TRACEFN(...);
   return DoRealizePalette(TRUE);
}
```

OnQueryNewPalette passes the buck to a helper function, DoRealizePalette, that does the work.

```
int CDIBView::DoRealizePalette(BOOL bForeground)
{
  CClientDC dc = this;
  CPalette* pPal = pDIB->GetPalette();
  CPalette* pOldPal = dc.SelectPalette(pPal, !bForeground);
  int nColorsChanged = dc.RealizePalette();
  if (nColorsChanged > 0) Invalidate(FALSE);   // repaint
  dc.SelectPalette(pOldPal, TRUE);
  return nColorsChanged;
}
```

DoRealizePalette gets the palette from the DIB and selects it in the foreground or background, depending on bForeground. In this case, bForeground is TRUE, so DoRealizePalette passes FALSE as the bForceBackground argument to SelectPalette. (It seems more natural to think of foreground as TRUE and background as FALSE, so I changed the sense of this flag.)

When DoRealizePalette calls dc.RealizePalette to finally realize the palette, Windows changes the system palette to match surf.dib's palette. As it turns out, Windows changes 235 colors in the system palette. But I don't know that yet because as soon as it does—before returning from ::RealizePalette—Windows immediately broadcasts WM_PALETTECHANGED to all top-level windows. Do not pass Go, do not collect $200. You can see this happening in **Figure 18**, where control jumps from DoRealizePalette to CMainFrame::OnPaletteChanged, before DoRealizePalette returns or even gets a chance to record how many colors changed. (This is where the TRACEFN

indenting helps reveal what's going on.)

As usual, MFC routes the WM_PALETTECHANGED to my handler.

```
void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
  TRACEFN("CMainFrame::OnPaletteChanged\n");
  const MSG& msg =
    AfxGetThreadState()->m_lastSentMsg;
  SendMessageToDescendants(WM_PALETTECHANGED,
    msg.wParam, msg.lParam);
}
```

CMainFrame broadcasts the message to all its children: "Hey, everybody, wake up, the palette just changed!" This time all the views get the message, not just the active view.

```
void CDIBView::OnPaletteChanged(CWnd* pFocusWnd)
{
  TRACEFN(...);
  if (pFocusWnd!=this)
    DoRealizePalette(FALSE);
  else
    TRACE0("[It's me, don't realize palette]\n");
}
```

The WPARAM argument of WM_PALETTECHANGED is the HWND of the window that changed the palette (in this case, surf.dib); MFC packages it as pFocusWnd. If the view is the same view that triggered the palette change in the first place (pFocusWnd == this), it ignores the message. After all, it's already realizing its palette in the foreground. In fact, right now it's in the middle of a call to ::RealizePalette, which is what triggered WM_PALETTECHANGED. Otherwise, if the view is not the view that changed the palette, it realizes its palette in the background by calling RealizePalette with FALSE. You can see all this happening in **Figure 18**: surf.dib reports "It's me, don't realize palette," whereas virgil.dib realizes its palette in the background, which results in 220 colors being remapped. Not 220 colors changed in the system palette, but 220 colors in virgil.dib's palette remapped to close-but-not-perfect colors in surf.dib's palette. As a result, the virgil.dib view invalidates itself so Windows will repaint it. If there were more views, they too would realize their palettes in the background and invalidate themselves if colors were remapped. If the WM_PALETTECHANGED had come as a result of the user activating some other completely different app like DOOM, then all the views, including surf.dib (even though it's the active view within DIBVIEW), would realize their palettes in the background.

Once all the views have handled WM_PALETTECHANGED, control returns to CMainFrame::OnPaletteChanged and back through Windows, back to ::RealizePalette and CDIBView::DoRealizePalette, where surf.dib realized its palette in the foreground in response to the original WM_QUERYNEWPALETTE. The result of surf.dib realizing its palette in the foreground is that 235 colors changed in the system palette. The surf.dib view invalidates itself and returns control back to CMainFrame::OnQueryNewPalette. The end. Well, actually the end doesn't come until Windows gets around to painting all the invalidated windows, but it will.

The second scenario is simpler. Here, DIBVIEW is already active, with surf.dib as the active view, when I click on virgil.dib to activate it. Now all the action takes place completely inside DIBVIEW, so Windows never sends a WM_QUERYNEWPALETTE message. (This is why I said MDI apps are a little more complex as far as

palettes are concerned.) It's up to you to handle the palettes in this situation. Fortunately, it's easy. All you have to do is realize the new view's palette in the foreground. The most convenient time to do it is when the view gets focus.

```
void CDIBView::OnSetFocus(CWnd* pOldWnd)
{
   TRACEFN("%s::OnSetFocus\n", DbgName(this));
   CView::OnSetFocus(pOldWnd);
   DoRealizePalette(TRUE);
}
```

Now when virgil.dib—or any other child frame in the MDI version of DIBVIEW—gets focus, it realizes its palette in the foreground, just as if the main window had received WM_QUERYNEWPALETTE. DoRealizePalette calls RealizePalette, Windows changes the palette and, assuming colors have changed, once again broadcasts WM_PALETTECHANGED. CMainFrame::OnPaletteChanged broadcasts the message to all the views, just as in scenario one. This time virgil.dib is the one to ignore it and surf.dib is the one to realize its palette in the background. surf.dib has 236 colors remapped as a result of virgil.dib changing 238 colors in the system palette (see **Figure 18**).

If you're totally confused, the best thing to do is download the code yourself and run it, or if you're on an airplane and the smelly guy next to you is hogging the phone, study the TRACE output in **Figure 18** and compare it with the code in **Figure 11**.

In the course of testing my palette code, I ran into a really nasty problem. Everything worked fine for the MDI version, which is what I primarily used to test with, because I just assumed MDI is harder. If it worked in MDI, it would naturally work in SDI. Wrong. For some reason, my palette code didn't work when opening a new file in the SDI version of DIBVIEW. A couple more TRACE statements revealed it was a bootstrapping problem. When you open a new file in SDI, MFC displays the File Open dialog. You select the file you want and press Open. As Windows destroys the dialog and activates DIBVIEW, it sends DIBVIEW a WM_QUERYNEWPALETTE message. But MFC hasn't loaded the new document yet, so you don't know what the palette is. The WM_QUERYNEWPALETTE message, your one chance in life to realize the palette, comes and goes before you even know what palette to realize. The fix was simple: just realize the palette in the foreground in OnInitalUpdate, after the document is opened.

```
void CDIBView::OnInitialUpdate()
{
                .
                .
                .

 DoRealizePalette(TRUE);
}
```

I realize that's a lot of palette-babble to digest in one sitting, especially if you've never seen palettes before. Sorry. But now DIBVIEW is something to be proud of. (Better than MFC's DIBLOOK, which takes 40 percent more source lines and can't even display 24-bit DIBs!) DIBVIEW displays all kinds of bitmaps, old and new, low and hi-color, and handles palette messages perfectly, a feat even some commercial apps don't get right. I tested DIBVIEW in several situations, with multiple instances running, overlapping windows, SDI and MDI, several DIBs open at once, 8- and 24-bit DIBs. In all cases DIBVIEW remained true to its, er, colors.

Figure 11 shows the full source code for step three of DIBVIEW. Don't forget to #include <vfw.h> in StdAfx.h for faster compiling, and don't forget to link with vfw.lib. VFW is a standard component of Windows 95 and the Visual C++ development tools.

Wrap Up

Uh-oh—I'm running out of space and time! Mr. Maximum Leader has already left three messages on my voice mail system, wanting to know when he can have the viewer. So excuse me while I run over to Acme World Headquarters to beg for yet another extension. Meanwhile, don't touch that dial! In Part II, I'll show you how to encapsulate all that hairy palette code in a totally reusable class, and I'll put the final touches on DIBVIEW by adding some neat features and more reusable goodies.

From the January 1997 issue of Microsoft Systems Journal. Get it at your local newsstand, or better yet, subscribe.