The Marriage of WinG and MFC

By Paul Modzelewski and György Grell

The Marriage of WinG and MFC

By Paul Modzelewski and György Grell

WinG, the new high-performance graphics library from Microsoft, brings to Windows 3.1 raw blitting speeds approaching those of MS-DOS. WinG is a very simple library, with one purpose, and one purpose only: fast blitting. All other functionality, such as sprites, must be added by you, the masochistic Windows programmer. Wouldn't it be nice to isolate all the WinG drudgery inside C++ classes and never think about it again? We thought so.

We set out to write some basic MFC classes to enable sprite based animation with WinG. A good starting point was the DOGGIE sample included with WinG. We extended its functionality by creating animated sprite objects, simplified creation of identity palettes with our own palette object, and created a class for handling DIBs. Our final application has an animated sprite (the word “WinG” rotating) which can be moved around with the mouse. Unlike Microsoft, we clean up after our “Doggie”, erasing each frame and leaving no trails.

We will cover basic WinG concepts lightly. For a detailed description of WinG you should refer to the help file that comes with WinG itself, which is readily available on both CompuServe and the internet (ftp.microsoft.com:/developr/drg/WinG/wing10.zip)

What is WinG?

WinG is simply an offscreen buffer which has functions to get that buffer to your screen as fast as possible. WinG contains ten functions, most of which manipulate the new device context, WinGDC, which works in tandem with the new bitmap, WinGBitmap, to form the buffer.

WinG allows you to get a pointer to the actual bits of the WinGBitmap selected into the WinGDC, and this ability makes the WinGDC/WinGBitmap combo very much like a Device Independent Bitmap (DIB). Once your image is composed in the offscreen buffer, you use WinGBitBlt or WinGStretchBlt to copy it to a display device context extremely rapidly. This is what WinG does well, and it does it very well. Unfortunately, that is the easy part.

The hard part is quickly composing your image in the offscreen buffer. Since the buffer is basically a DIB, what you need is a way to copy all or some of a DIB into another DIB. You could use standard Windows GDI calls to do this, but GDI's convenient device-independence is also painfully slow when it comes to blitting DIBs. This means you will

have to write your own blitting routine, which is basically a fancy memcpy. Depending on the needs of your application, you may also want to write your own fast drawing routines (i.e. polygons, texture mapping, fractal drawing, etc.)

What we did

We considered several designs, and decided that WinG itself doesn't need to be wrapped into a class because it is so simple. The main difficulties in using WinG are palette control and DIB manipulation. We built the CWinGPalette and CDib classes to handle these issues. In addition, we created CSprite and CPhasedSprite to manage movement and animation of sprites.

Since we didn't need the extra overhead of the document/view architecture, we implemented all of our application's work in the CMainFrame class. This includes setting up WinG, responding to window messages, drawing, and data storage.

Just before we wrote this article, Microsoft released yet another tool, WinToon. WinToon is a library that uses WinG and Video for Windows to do some neat tricks with animation. However, included with WinToon is a wonderful set of 32-bit assembler blitting routines, and a bunch of nice DIB management routines as a bonus (WinToon is available at ftp.microsoft.com:/developr/drg/Multimedia/WinToon/wintoon.zip).

Being "efficient", we decided to use WinToon's DIB routines. These are the base of CDib's functionality. In addition, the 32-bit functions are used in the CSprite class. All these functions are nicely packaged into llibdib.lib (Hey, we didn't name it!) The DIB management function prototypes are in dib.h, and the 32-bit assembler function prototypes are in dibfx.h. There is other cool stuff in there that we didn't even touch. Check it out.

CWinGPalette

We soon realized that the first thing to tackle was identity palettes. The identity palette concept is probably the most important and difficult part of understanding and using WinG. An identity palette is a Windows palette object that exactly matches the hardware palette in your video card. If you don't have an identity palette, WinG will have to translate every pixel in your image as it displays it. This is stunningly slow.

To get an identity palette in WinG, you must make sure that the color table in your WinGBitmap and the palette selected into the display device context you are drawing on both match the hardware palette

exactly. This can be a tedious affair, at best, and is nicely hidden in CWinGPalette.

CWinGPalette descends from MFC's CPalette class, and adds all the functionality required to set up an identity palette from a series of bitmaps. WinG only supports eight bits per pixel, therefore we only have 256 colors to work with. Windows has twenty static colors that it uses for titlebars and such. Since we do not want to change these colors, we will need to put these into our own palette. In CWinGPalette's constructor, the twenty static system colors are grabbed from the system, and stored in the internal palette structure. This means we only have 236 colors available for our own use.

CWinGPalette has only four methods:

```
int  AddQuad(LPRGBQUAD pRgbQuad, BOOL bCollapse = TRUE);
void AddColors( CDib& dib, BOOL bScanPixels, BOOL bCollapse = TRUE );
void CopyQuadsTo( LPRGBQUAD lpRgbQuads, int nNumColors = 256 );
void CreatePalette();
```

AddQuad lets you add a single RGBQUAD to the palette. An RGBQUAD is a standard Windows structure which is four bytes long. It contains a byte each for red, green, and blue color values, and an additional reserved byte. When you add an RGBQUAD, the function scans the existing colors to make sure there is no duplication. Since we are using the PC_NOCOLLAPSE flag to make sure Windows doesn't remove duplicates for us, we do it ourselves in a controlled manner. Sometimes you may not want to eliminate duplicates from your palette. Setting the bCollapse parameter to FALSE will prevent AddQuad from scanning for duplicate colors. If successful, AddQuad returns the palette index of the color you are attempting to add, otherwise it will return ERR_PALETTEFULL (our own error value). If your new color already exists and bCollapse is TRUE, AddQuad will not add the color, but it will return the palette index that matched your new entry.

The AddColors function operates in one of two ways, depending on the value of the bScanPixels parameter. When bScanPixels is FALSE all the colors in the DIB's palette are copied into the CWinGPalette, dropping any values after reaching 236 colors. If bScanPixels is TRUE, we don't trust the DIB's header and scan each of the DIB's pixels to see which colors are actually used. Checking every pixel is a serious performance hit, but we decided to add this functionality since it would be handy in a palette editing program. Generally, in a game or other high performance program, you would load a precreated palette from a file.

CopyQuadsTo simply copies the requested number of colors from the palette to an array of RGBQUADs. This is the function we use to set the palette of the WinGBitmap once we have created our application's palette.

CreatePalette is an override of MFC's CPalette::CreatePalette (thank Microsoft for not making this function virtual) and creates the Windows palette object. We use this function in CMainFrame during the setup of WinG.

CDib

Next up is the CDib class. MFC does not contain any classes to aid in the management of DIBs, so we wrote our own. CDib descends from CObject (the ancestor of almost all MFC classes), and uses the DIB management routines from WinToon as the base of its functionality. Included are member functions to load a DIB, get the width and height of the DIB and access the bitmap's bits. The ones we will discuss are:

```
BOOL Load( LPSTR fileName );
void MapToPalette( CPalette& palette );
```

The DIBs can be stored either in files or as resources. Load will assume that the string passed to it identifies a file. If it can't find that file, it will try to load a resource with that string name. This is a feature that came free with the WinToon DIB functions.

Figure 1

Now that we have some DIBs loaded into memory, and a palette created from them, we need to make sure the bitmaps' colors match the palette's. This is accomplished with the MapToPalette function. This method gets the colors from the palette, and matches the bitmap's colors to the palette's, as illustrated in Figure 1. This way you won't end up having an oddly colored picture.

CSprite and CPhasedSprite

CSprite descends from CObject, and is used to store both position information and a CDib pointer. The position information is stored in the form of two CRects, m_rect and m_prevRect. The sprites latest position is stored in m_rect, and its previous position is stored in m_prevRect.

```
void DrawSprite( LPBITMAPINFOHEADER lpBIH, LPBYTE lpByte, int nTransColor );
void EraseSprite( LPBITMAPINFOHEADER lpBIH, LPBYTE lpByte, CDib& background  );
```

The DrawSprite member function of CSprite uses the DibTranparentBlt function from WinToon to draw the sprite's bitmap. The bitmap will be drawn into the DIB bits identified by lpByte (usually your WinGBitmap)

in the position marked by m_rect. All pixels with the value of the nTransColor parameter will be ignored by DrawSprite, making these pixels “transparent”, since they will not show up in the destination.

EraseSprite works much like DrawSprite, with two major differences. First, instead of drawing the sprites bitmap in m_rect’s location, it will draw with the CDib indicated by the background parameter in m_prevRect’s location. Second, it does not deal with transparency at all and uses DibBlt instead of DibTransparentBlt.

Descending from CSprite is CPhasedSprite. A CPhasedSprite is a sprite with different phases, basically, it is a frame animated sprite. CPhasedSprite differs from CSprite by storing an array of pointers to DIBs, and exposing member functions that allow changing between frames.

Setting up WinG

The following is our WinG setup function in CMainFrame. The first step in setting up WinG is calling WinGRecommendDIBFormat. When the WinG DLL is loaded for the first time on a particular display setting, it will do extensive profiling of your display hardware, determining the fastest DIB format on your machine. Calling WinGRecommendDIBFormat will return this format.

```
//  Get WinG to recommend the fastest DIB format
if( WinGRecommendDIBFormat( (BITMAPINFO*)&m_bufferHeader ))
{
  //  make sure it's 8bpp and uncompressed
  m_bufferHeader.bih.biBitCount = 8;
  m_bufferHeader.bih.biCompression = BI_RGB;
}
else
{
  //  set it up ourselves since nothing was recommended
  m_bufferHeader.bih.biSize = sizeof(BITMAPINFOHEADER);
  m_bufferHeader.bih.biPlanes = 1;
  m_bufferHeader.bih.biBitCount = 8;
  m_bufferHeader.bih.biCompression = BI_RGB;
  m_bufferHeader.bih.biSizeImage = 0;
  m_bufferHeader.bih.biClrUsed = 0;
  m_bufferHeader.bih.biClrImportant = 0;
}
m_bufferHeader.bih.biWidth = m_nDispWidth;
// Retain the orientation
m_bufferHeader.bih.biHeight *= m_nDispHeight;
```

If WinGRecommendDIBFormat returns FALSE, we do our own setup of the DIB format.
We’ve never seen this happen, but you never know. The next step in our setup function is to prepare and create our palette. The first step in creating our palette is to call the CWinGPalette function AddColors for every bitmap we intend to use. This will fill the palette with all the used colors in each bitmap (provided there are less than 236 total colors). After we create the palette, we remap all the bitmaps’ colors

to this new palette. See Figure 1 for more information on the remapping process.

```
// Load the palette with colors from bitmaps, then create it
m_palette.AddColors( *(CDib*)m_dibs[0], TRUE );
m_palette.AddColors( m_background, TRUE );
m_palette.CreatePalette();

// Remap the existing bitmaps' colors to the new palette
m_background.MapToPalette( m_palette );
for( int nIndex=0; nIndex < 30; nIndex++ )
  ((CDib*)m_dibs[nIndex])->MapToPalette( m_palette );
```

Now that we have created a palette from the colors of all our bitmaps, we use the CopyQuadsTo function of CWinGPalette to copy those colors into the header of a WinGBitmap. This is how we create an identity palette, letting CWinGPalette do all the work for us. Once m_palette is realized, the system palette, our application's logical palette, and the color table of our WinGBitmap will all match, giving us an identity palette.

```
// Fill the WinGBitmap's header with the colors
m_palette.CopyQuadsTo( m_bufferHeader.aColors );
```

The final few steps of setting up WinG are the creation of a WinGDC and a WinGBitmap, and selecting the bitmap into the DC, like so...

```
//  Create a WinGDC and Bitmap
m_hWinGDC = WinGCreateDC();
HBITMAP hBitmap = WinGCreateBitmap(
  m_hWinGDC,
  (BITMAPINFO*)&m_bufferHeader,
  &m_pDispBuffer );

//  Store the old bitmap to select back in before deleting
m_hOldBitmap = (HBITMAP)::SelectObject( m_hWinGDC, hBitmap );
```

You'll notice that we store the original bitmap contained in the WinGDC, so we can restore it before we quit our app. It is important to remember that a WinGDC is still a device context. This means you can use GDI to write on it, and it follows most of the same rules as every other DC.

A day in the life of CMainFrame

Okay, now that WinG is all set, what to do with it? The first thing we have to do is select and realize our CWinGPalette, m_palette, into our display device context, m_pDC. It is important to mention that we hold m_pDC for the life of our application, creating it in CMainFrame::OnCreate, and destroying it in CMainFrame's destructor. This way we won't need to create a new DC object each time we use WinGBitBlt.

The selection and realization of our palette happens in OnSetFocus in CMainFrame. When we lose focus, another application could change

the palette, so this will ensure that we always have an identity palette when we are the active app.

```
void CMainFrame::OnSetFocus(CWnd* pOldWnd)
{
  CFrameWnd::OnSetFocus(pOldWnd);

  m_pDC->SelectPalette( &m_palette, FALSE );
  m_pDC->RealizePalette();

  // start the timer
  m_nTimerID = 1000;
  SetTimer( m_nTimerID, 1, NULL );
}
```

The last thing we do in OnSetFocus is start the standard Windows timer. The standard Windows timer is only good for about 18 ticks per second. If you want a higher resolution timer you'll have to use the Windows multimedia timer services. We stop the timer in OnKillFocus. We used Class Wizard to create the function OnTimer, which is called every time the timer ticks.

```
void CMainFrame::OnTimer(UINT nIDEvent)
{
  // Step to its next frame
  m_pSprite->NextFrame();

  // Clear the old area of the sprite with the background bitmap
  m_pSprite->EraseSprite(
    (LPBITMAPINFOHEADER)&m_bufferHeader,
    (LPBYTE)m_pDispBuffer,
    m_background );

  // Draw the sprite in the new position
  m_pSprite->DrawSprite(
    (LPBITMAPINFOHEADER)&m_bufferHeader,
    (LPBYTE)m_pDispBuffer,
    0 );

  // Take the union of the previous rect and the new rect and blit that
  // area onto the window
  CRect updateRect;
  updateRect.UnionRect( m_pSprite->GetPrevRect(), m_pSprite->GetRect() );
  WinGBitBlt(
    m_pDC->GetSafeHdc(),
    m_nXOffset + updateRect.left,
    m_nYOffset + updateRect.top,
    updateRect.Width(), updateRect.Height(),
    m_hWinGDC,
    updateRect.left, updateRect.top);

  m_pSprite->SetPrevRect();
}
```

This function is the heart of our application. You can see that the first thing we do is call CPhasedSprite::NextFrame, which sets the sprite's DIB to the next in the series. The next step is to redraw the background in the position where the sprite used to be, by calling EraseSprite. Once the sprite is erased, we redraw it in its new position with DrawSprite.

Figure 2

It is important to understand that at this point we still haven't copied anything to the screen, yet. All the blitting we have done so far was onto our offscreen buffer, the WinGDC. Performing image composition in an offscreen buffer, and then blitting the finished image onto the screen reduces flicker and provides smoother animation. This is called "double buffering", and is illustrated in Figure 2.

To get these images onto the screen, we'll use WinGBitBlt. We could simply blit the entire offscreen buffer onto the display every time we moved or changed the sprite, but this would be extremely inefficient. To minimize the amount of data we must send to the screen, we will union the sprites current and previous rectangles, giving us the smallest possible rectangle that must be updated.

We do basically the same thing in OnMouseMove that we do in OnTimer. When the left mouse button is clicked, CMainFrame will check to see if the hit fell within the bounds of the sprite. If it does, moving the mouse around causes the sprite to move, until the left button is released. You can use a right double-click or the escape button to exit the application.

Wrapping Up

As we have shown, using MFC and high-performance graphics are not mutually exclusive. It is even possible to get frame rates fast enough for arcade games using modified versions of the code we have supplied. The biggest obstacles to high-performance C++ are constructors. Be aware of when your compiler will silently call them, and learn how to avoid those situations.

We have tried to make our classes as extensible as possible, and we feel they are a good start at a WinG class library. A "good start" is a long way from complete, though, and there are many specific things that you may want to add. At the very least, reading and understanding all of our source code should give you a firm grasp of WinG fundamentals, and some decent tools to help you along your way.

Paul Modzelewski *works for Magnet Interactive Studios, where he specializes in Windows multimedia and 3DO game programming in C++. He can be reached at pmodz@magnet.com .*

Gyorgy Grell *works for MLJ, Inc., where he specializes in Windows GDI and GUI programming in MFC/C++ and SDK/C. He can be reached at gsoft@netcom.com*