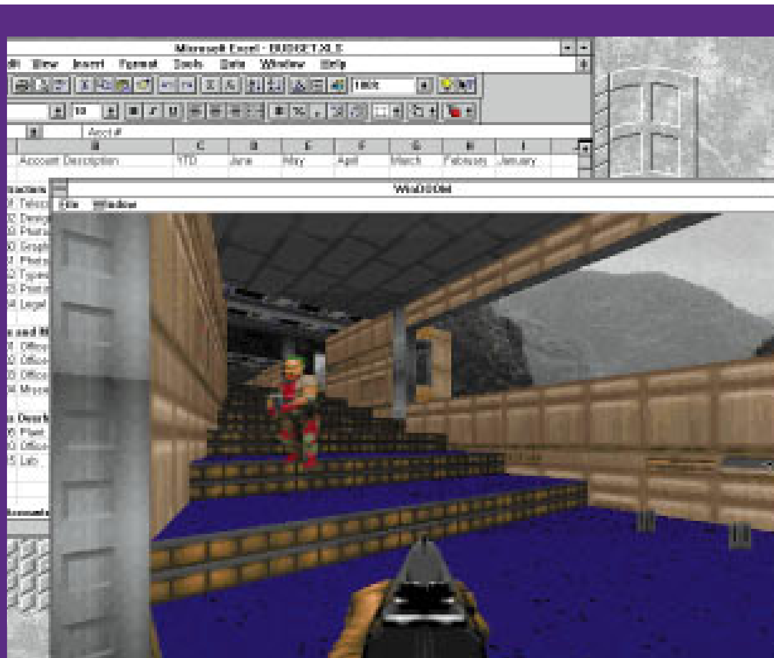
Attention:

Thank you, and I hope you enjoy the article,

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

A Whirlwind Tour of WinG



When WinDoom was ported, the rendering vs. stretching issue was left to the user by providing a menu of choices. Now you can set it to render to the current window size or preset the size and stretch to the current window size.

If you're like me, the first time you saw Microsoft Windows 3.0 and its program manager, you went straight for the Games program group. Like me, you probably expected to find a game as different from DOS games as Windows is different from DOS itself. Instead, you found Solitaire. Not a bad version of Solitaire, but Solitaire nonetheless. If you waited until 3.1 to check out Windows, you also found Minesweeper—a bit more exciting, but you wouldn't call it "high-performance."

Expectations for Windows games have been very low. When Microsoft released a set of games called Arcade last year, reviewers were shocked. They couldn't believe games of Arcade's quality could be done on Windows. Arcade is a great set of games, but we are talking about 1970s technology on 1990s computers! Their enthusiasm was unfounded: Arcade is nothing compared to the games you find on DOS. A Pentium probably has more on-chip cache than the original Asteroids game had main memory.

Sure, operating systems of today do more than they did back then (did they even have operating systems back then?), and I can play Asteroids while simultaneously running other applications on the same desktop, but is this all we can expect from our brand new machines running Windows? On the same hardware, DOS games have consistently pushed the performance envelope with the current crop doing full-screen texture-mapped worlds at 30 frames per second. What's the crucial difference between DOS games and Windows games? Graphics performance.

Finally there's help: WinG. WinG is a library that eliminates the performance difference between DOS and Windows graphics, giving Windows games graphics performance at or above their DOS counterparts on the same hardware.

Current Windows Graphics—Slow?

We're interested in raw blt (bit level transfer) performance: transferring pixels to the screen in blocks. Most high-performance games try to achieve smooth animation by hiding the rendering and only allow the player to see the resulting frame. These games compose images into buffers, then quickly update the display. While the composition phase is usually application-by Chris Hecker

Does graphics performance set DOS and Windows a world apart? Think again. Because of WinG, Window's graphics are flying high, giving performance at or above their DOS counterparts.

specific (each game renders using its own special algorithms), only a few popular techniques for updating the display exist.

Update techniques fall into two groups: blting and page flipping. The trade-offs between the two techniques on current PC hardware are far too complex to cover here, but suffice it to say that high-performance DOS games use both techniques (for example, System Shock and Ultima Underworld I and II blt, while Doom page flips). It is fairly easy to move a game from blting to page flipping or vice versa.

Windows does not currently allow page flipping, so we will deal with blt performance. Although we've said graphics speed (or lack thereof) is the major impediment to high-performance Windows games, if you time the `BitBlt` function, you will find the bandwidth comparable to what you find under DOS for the same resolutions. The catch is `BitBlt` transfers pixels from objects called `HBITMAP`s, not from memory the application owns.

Applications are not allowed to touch the bits of an `HBITMAP` directly, they must use Windows Graphics Device Interface (GDI) functions, like `LineTo`, `SetPixel`, and `Rectangle`. GDI provides a rich set of two-dimensional graphics functions that are perfect for applications like spreadsheets and word processors, but you will not find a `TextureMapPolygon` function anywhere in the Windows API documentation. For this reason, games need to render directly to memory, and GDI does not allow them this luxury with `HBITMAP`s.

Windows does provide objects called Device Independent Bitmaps (DIBs), which applications can access directly, but the APIs for transferring DIBs to the screen (`StretchDIBits` and `SetDIBitsToDevice`) are typically three to 20 times slower than `BitBlt` and therefore not competitive with DOS blt bandwidth.

WinGBitmaps—a Hybrid

WinG introduces a new kind of object: the `WinGBitmap`. `WinGBitmap`s are both DIBs and `HBITMAP`s. Applications get a pointer to the bits like a DIB, and like an `HBITMAP`, WinG will transfer them to the screen quickly. How quickly? At the 1994 Game Developer's Conference, we demonstrated a Windows version of Doom, WinDoom, running at about the same speed under Windows as the DOS version on the same hardware. Better yet, it only took a weekend to do the port.

Porting a DOS Game to WinG

I don't have space in this article to develop a DOS game and then port it to Windows and WinG, but I will describe a typical DOS game's architecture and discuss how to move it to WinG. Let's assume our game has five major parts:

- Setup
- Get input events
- Run the simulation
- Render into a buffer
- Blt the buffer to the screen.

During `Setup`, the program allocates the off-screen buffer, creates the palette, and initializes the simulator. Next, it gets any user input and uses that information to run the simulator for a single time slice. The results of the simulation are rendered into a buffer, and the buffer is blted to the screen. We're ignoring synchronization, sound, networking, user interface, and whatnot, but you get the idea.

Under Windows, the setup phase needs to initialize Windows-specific elements, like the application window, but most of the setup code stays the same. One interesting difference is that, unlike the DOS version where your application allocates the buffer memory, you must call `WinGCreateBitmap` with `BITMAPINFO` (a structure describing the size and format of the `WinGBitmap`) to allocate the buffer, and WinG will return the memory pointer. The application uses this pointer to draw on the `WinGBitmap` surface directly.

The application will also need to use GDI palette APIs to create and realize the game's palette. GDI realizes a palette when it copies the description of the palette colors into the video hardware. Because multiple applications can share the hardware palette, this can get a bit tricky, but there is plenty of palette sample code in the WinG development kit to illuminate matters.

User input is slightly more difficult. Well-behaved Windows applications must yield control to the system fairly often in case the user wants to switch away to another application. Normal applications like word processors call the `GetMessage` API to process their user input messages. If there are no messages for the application, `GetMessage` doesn't return until one comes in.

A game can't use `GetMessage` because even if the player isn't providing input to the application, the simulation must still run. You don't want the whole game to stop when the user stops pushing keys or moving the mouse, so Windows provides an API called `PeekMessage`. This API returns immediately even if there are no messages so the game can continue the simulation. The subtleties of `PeekMessage` in particular and event-driven architectures in general are beyond the scope of this article, but I will provide you with an appropriate reference.

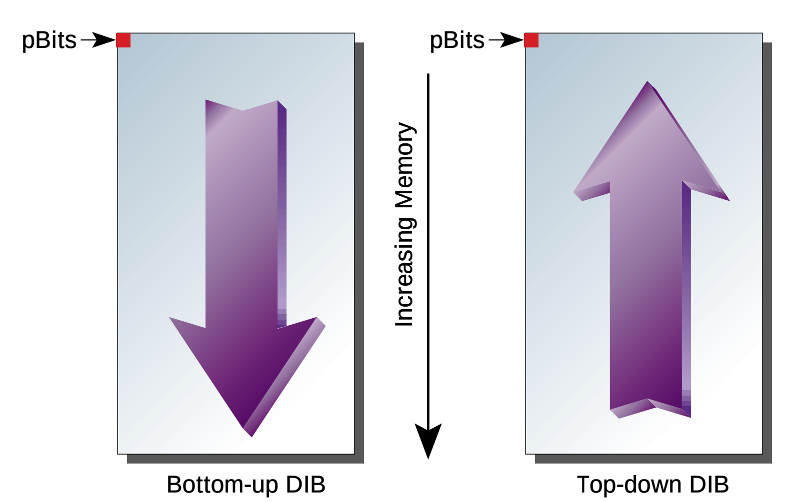
The game simulation code should work unchanged on Windows. Once the user input is translated from Windows messages to the application-specific format, the simulation should run normally.

Your game's rendering code should also work unchanged. The only caveat is that `WinGBitmap` scanlines are `dword` aligned, so if for some reason you need a 201-wide bitmap, you'll need to know the start of the next scanline is actually 204 bytes from the current scanline, not 201 bytes.

Once composition is complete, you blt the buffer to the screen with `WinGBitBlt` or `WinGStretchBlt`. As its name implies, `WinGStretchBlt` will stretch or compress the `WinGBitmap` as it blts, where `WinGBitBlt` simply transfers the `WinGBitmap` to the screen.

Once you have your game running on Windows, it's time to make it run fast. You'll also want to take advantage of the benefits of running in a windowed environment, so we'll talk about some of those issues as well.

Figure 1. DIB Orientations



Setup

Our naive port called `WinGCreateBitmap` with the description of the `WinGBitmap` we wanted. To achieve maximum blt performance during the screen update phase, we'll ask WinG for a little help during our optimized setup. Although WinG is fast under almost any circumstances, there will always be a particular `WinGBitmap` format that is the absolute fastest to blt on the current display, and the `WinGRecommendDIBFormat` API will tell us what that format is at run time.

The most important difference between the DIB formats that we'll get back from `WinGRecommendDIBFormat` is the DIB orientation. There are two DIB orientations: bottom-up and top-down, illustrated in Figure 1. Both kinds of DIBs consist of a `BITMAPINFO` structure and a pointer to the bits. The `BITMAPINFO` contains information such as the width, height, number of bits per pixel, and the color table of the DIB. For bottom-up DIBs, the bits pointer points to the bottom-most scanline in the DIB.

Increasing memory addresses means going up the DIB image, hence the term bottom-up. This is probably the exact opposite of the memory bitmaps you've dealt with before and is the opposite of most video displays (notably mode 13h VGA, for example). Top-down DIBs are more familiar: the bits pointer points to the top-most scanline, and increasing memory addresses go down the image. Life gets interesting because WinG might recommend either DIB format at run-time, and high-performance games should be able to deal with both. This isn't as hard as it sounds. I'll go over the details in the section on rendering.

Once we have the recommended DIB format, we pass the information to `WinGCreateBitmap` and go on to our palette setup. For optimal performance, a WinG application should have an "identity palette mapping." An identity palette mapping means the color table in the `WinGBitmap` and the palette in the display hardware match exactly. In this case, WinG can block-transfer the pixels in the `WinGBitmap` to the screen without translating them. If the palette mapping is not identity, WinG needs to translate each pixel as it is blted, which is slow. We'll cover this briefly, and if you still don't get it, there is plenty of excruciatingly detailed documentation and sample code in the WinG development kit.

Windows runs multiple applications at the same time. There is only one hardware palette—something has to give. The compromise is that each application requests the hardware palette (called the system palette) by calling `RealizePalette`. Windows may or may not let the application have the entire system palette depending on a number of factors, like whether the application is in the foreground, whether there are other palette applications around, and so on.

Even if Windows does give the application the system palette, the system tries to minimize the palette entries used by each application by collapsing any duplicate colors into the first instance of that color. In addition, each `WinGBitmap` has an application-defined color table associated with it, and the color table must match the system palette for the mapping to be identity. If all this sounds complicated, it is, but once you understand it, you'll be able to charge outlandish consulting fees to other game developers, so it's worth your time to learn. Besides, your blts will go from mediocre to blazing once you get an identity palette mapping.

WinG can help in your quest for an identity mapping by spitting out debugging information. You can set two flags in the win.ini configuration file to direct WinG to tell you what is going on. The `Debug` flag makes WinG tell you if you have an identity palette mapping, and the `DebugPalette` flag makes it tell you how each color table index in your `WinGBitmap` maps to the current system palette if that mapping is not identity. So, if you can't figure out why you don't have an identity palette, you can turn on `DebugPalette` and see messages like:

```
WinG: Palette mapping is not identity.
WinG: Color table index 123 maps to
      system palette entry 5.
```

You can take this information and see exactly why you aren't getting an identity mapping.

As soon as you've figured out the intricacies of identity palettes, you'll need to make a user interface decision: `SYSPAL_STATIC` mode or `SYSPAL_NOSTATIC` mode. Windows normally reserves 20 colors in the system palette and does not let applications overwrite them. This keeps a single palette application from making all other applications look horrible—other applications always have at least those 20 colors, called the static colors, to map to, even if an application realizes an all-black palette. As with most things in Windows, there's a way around the static colors: `SetSystemPaletteUse`. If you call `SetSystemPaletteUse` with `SYSPAL_NOSTATIC`, Windows will let you overwrite 18 of the 20 static colors, leaving only black at entry 0 and white at entry 255.

`SYSPAL_NOSTATIC` applications make the Windows desktop look gross, while `SYSPAL_STATIC` applications only get 236 colors out of a possible 256. You'll need to choose which mode to use as you develop your game. It is possible to use SYSPAL_NOSTATIC when you have a maximized window (users won't be able to see the off-colored desktop anyway) and SYSPAL_STATIC when you're windowed (and users can see the program manager and other applications), but your game must do the extra work.

Rendering

High-performance games have optimized rendering algorithms, and most of this code can be left alone, although your rendering code will need to deal with top-down and bottom-up DIBs for best performance. The impact this has on most rendering code is minimal. When you step from scanline to scanline, you need to use a signed number. For example, let's say this is your rendering loop for a 320 byte wide buffer:

```
   ; edi points to destination scanline
   mov    edi,pBits
loop_top:
   ; draw some pixels
   mov    [edi],ThisValue
   mov    [edi+4],ThatValue
   mov    [edi+8],TheOtherValue
   add    edi,320   ; point to ext
                      scanline
   dec    ScansLeft ; if we're not done,
   jnz    loop_top  ; do it again
```

Although simple, this type of loop is the core of most scanline renderers. After changing two lines, this code can handle both DIB orientations at run time:

```
mov   edi,pBits
```

becomes:

```
mov   edi,pTopScanline
```

where pTopScanline is the first scanline (pBits) on top-down DIBs and the last scanline (pBits + WidthInBytes * (Height - 1)) on bottom-up DIBs. The second change is:

```
add   edi, 320
```

to:

```
add   edi,DeltaScan
```

where DeltaScan is 320 for top-down DIBs and -320 for bottom-up DIBs. This change causes the renderer to always move down the image, increasing edi for top-down and decreasing it for bottom-up.

A second issue affecting the renderer is variable-sized viewports. Because Windows runs at whatever resolution the user chooses, games should be able to handle different window sizes. There are two ways to do this: the game can render at different resolutions, or the game can use WinGStretchBlt to stretch a constant-sized buffer to the variable-sized window. The former is a rendering issue, the latter affects the blt/update code as well.

Blting

There are tradeoffs between rendering at the viewport resolution and calling WinGBitBlt to blt the buffer and rendering at a lower resolution and calling WinGStretchBlt to expand the buffer to the viewport resolution. If your renderer can handle high-resolution buffers, you'll get the best looking results by rendering at the resolution of the viewport, but you might find the performance is too slow. If your game is pixel-bound, like Doom (in other words, it spends more time rendering a pixel than WinG spends blting or stretching that pixel), you may want to take advantage of the high-performance stretch code in WinGStretchBlt, render to a low resolution buffer, and stretch it to fill the viewport.

When we ported WinDoom, we left the rendering vs. stretching issue to the user by providing a menu of choices. You can set it to render to the current window size (which really slowed down as the window got larger), or it could render at a preset size and stretch to the current window size. WinGStretchBlt is extremely fast, so the stretching option usually resulted in the best frame rate, but it didn't look as nice as the full rendered version. Most DOS games have level-of-detail settings, so users can choose stretching versus rendering as they like.

Other Issues

It's been said that the best and worst thing about Windows is that it runs on an incredible variety of hardware. To make the most of this variety, your game will need to configure itself to the run-time platform, like WinG does at startup with the display performance test. Is it faster to stretch or render? The answer will change depending on the user's hardware and software configuration, so be prepared. Is it faster to update dirty rectangles or blt the whole buffer? Again, this can change from machine to machine. Time it and you'll never go wrong.

This has been a whirlwind tour of WinG game development, but we've touched on the major issues. Once you are seriously into Windows programming, get the WinG development kit for yourself and play with the sample applications to get first-hand experience, then port your game to Windows in no time flat. ■

Chris Hecker works for a large software company in the Pacific Northwest. He can't mention the name because then he'll need all sorts of disclaimers. It's just a coincidence that he can be reached at checker@microsoft.com. or through Game Developer *magazine.*

WINGING IT

Is it time you port your own game to Windows? Then it's time to start using WinG. For the satisfaction of porting your own game in no time flat, consult these resources. Don't sit around while someone does it for you—WinG it!

FOR BEGINNERS:
Programming Windows 3.1 by Charles Petzold (Microsoft Press, 1992)

FOR EXPERIENCED DEVELOPERS:
The Microsoft Developer's Network CD-ROM (Microsoft Developer Network, (800) 759-5474).

FOR EVERYONE:
Microsoft's WinG Development Kit is available from the winmm forum on CompuServe or on ftp.microsoft.com