Writing HOT Games for Microsoft® Windows™

The Microsoft Game Developers' Handbook

Microsoft Windows™ Multimedia

4/94

Table of Contents

MICROSOFT WINDOWS MULTIMEDIA

Microsoft is committed to making Windows a leading force in multimedia technologies and systems.

Our commitment takes many forms, but the most important one for independent software vendors is our commitment to substantial and ongoing investment in multimedia-related R&D. In this handbook we explain some new innovations in Windows that will be of particular interest to game developers.

We're serious about games

For the past year, the home market has been the fastest-growing segment of the PC business. More and more of our customers are telling us that they want games for Windows − and at this point, there aren't many. Games are already the largest category of multimedia application, but most of today's computer games are running on MS-DOS®. In fact, at the end of 1993 computer games were one of the last remaining software categories for which Windows product sales trailed MS-DOS product sales.

Not that this should come as any surprise. Until now, game graphics under Windows made slug racing look exciting.

The only way for Windows to succeed as a game platform is for developers to write great games for it. This Handbook is designed to help you do that. And the WinG library delivers graphics performance approaching raw MS-DOS speeds.

Millions of new 32-bit game machines on the way!

There are over 100 million MS-DOS based personal computers in the world − and over 40 million of those are running Windows. The home PC market is growing fast, and a very large portion of the machines being sold into homes are equipped with the kinds of devices that should make game developers smile: CD-ROM drives, sound subsystems, and 4MB of RAM or more.

With the release of Chicago, a GIANT new 32-bit game platform will be born.

This handbook:

1 explains some of the new features and capabilities of Chicago that make it possible for you to write great games for the world's next PC operating system; and
2 introduces the WinG libraries, a development tool that you can use to write high-performance graphical games for Windows today.

It ain't over yet

It's important to emphasize that the technologies described in this handbook aren't the end of the story. Quite the contrary − we've made some very important first steps, but there's a lot of work to be done in the years ahead. Please tell us how we can make Windows a great platform for your games!

We've invested a lot of effort making Windows into the leading environment for "serious" applications.

Now comes the fun part!

To send a suggestion to the Microsoft Windows Multimedia team, GO WINMM in CompuServe® or send email to mmdinfo@microsoft.com.

THE WINDOWS MARKET

It has now been over four years since the release of Windows 3.0, and the software market is greatly changed. For example:

- The majority of new PCs sold now come with Microsoft Windows pre-installed.
- Virtually every PC peripheral and add-in board on the market (including sound cards) ships with drivers and installation instructions for use with Microsoft Windows.
- In virtually every software category (with the exception of games), sales of software for Windows outpace sales of software for MS-DOS.

Windows is a growing, active platform, and coming developments will help to make it more so. “Chicago” is the code name for the next version of Windows. Analysts who have reviewed Chicago in its prerelease form are already concluding that this new operating system will be the biggest news to come from Microsoft since the release of Windows 3.0.

Until recently, the Windows-based personal computer hardware and software businesses focused almost exclusively on the office desktop. That focus is broadening rapidly:

- Home PCs are the fastest-growing category of the PC hardware business, and home buyers are demanding multimedia capabilities in their PCs.
- Laptop and notebook computers are an increasingly important share of the business, and they are helping to establish the modem as a standard PC device.
- Non-business software categories (including games) are rapidly growing into the multi-billion dollar range.

These trends show no signs of stopping. As PCs become an increasingly common appliance in the home, we can count on the following corollaries:

- The average level of computer knowledge among home users will decline.
- The market will favor ease of installation and ease of use as features of home software products. Products that are obnoxious or scary to install won’t sell well in the broad home market.

These forces will help support the continued success of Windows software in the home market, which presents an important opportunity for the game development community. There are millions of PCs with Windows in homes today, and relatively few Windows-based games. The market forces above make it pretty clear that there’s a big market out there, waiting for the right Windows-based game products to come along.

Got any ideas?

Chicago as a Game Platform

This chapter describes some (not all!) of the features of Chicago that will make it an important release for you as a game developer. In the case of our work on graphics speed, this chapter also describes how WinG makes it possible for games to smoothly transition to Chicago while maintaining compatibility with Windows 3.1 today.

This document is an overview of our technology and development plans as they relate to games. Further technical articles about implementing games on Windows can be found in the Microsoft Developer Network. (For a subscription, call Microsoft at 1-800-759-5474. Outside the US and Canada, call 402-691-0173 for a distributor in your country.)

Naturally, we are interested in your input in all of these areas, as we are very committed to making Windows the best game operating system possible. The best way to reach us is through the new CompuServe forum for game developers – GO WINMM. Or send email to mmdinfo@microsoft.com.

Discussed here are the following topics:

1 Fast, Flexible Graphics
2 Easy to Support
3 Powerful Development Environment
4 Built-in Digital Video Support
5 High Quality, Flexible Sound
6 Support for Multi-player Games
7 Synchronization
8 Toys
9 3D for Windows

1 Fast, Flexible Graphics

The speed of graphics (or, more appropriately, the lack of it) in Windows 3.1 has been the most important obstacle keeping game developers from choosing the Windows platform for their games. We have addressed this issue head-on in a way that provides substantially improved speed while preserving the device independence that makes Windows appealing in the first place with a new graphic library called WinG.

WinG provides close to the graphic performance of MS-DOS based games with all of the device independence of Windows. (For detailed information about the WinG library, see Appendix A)

WinG overview:

Fast graphics. Blts directly to the frame buffer from Device Independent Bitmaps in memory, allowing performance approaching raw MS-DOS speed on any given Windows compatible computer.

Compatible with Windows 3.1 and Windows NT. Runs across Microsoft operating systems, *including the millions of copies of Windows 3.1 installed today*, and uses the fastest available blting mechanism on each OS.

Color. WinG is optimized for all color depths and graphics resolutions under Chicago and Windows NT. For WinG on Windows 3.1, MS-DOS-class graphic performance is optimized for 256-color source animation.

Free. The WinG library will be freely available to developers and redistributable at no cost. This library will be available on CompuServe in the WINMM forum by the end of May, 1994.

32 bit. WinG will be available in both 16- and 32-bit versions, allowing you to create full Win32 games.

Other graphic enhancements

Chicago games will also be able to take advantage of the following enhancements to the operating system:

On-the-fly control of screen resolution makes it possible to dynamically configure the display to achieve optimal speed and graphics quality for your game.
Offscreen buffering and other hardware-level video functionality not accessible in Windows 3.1 will also be supported in Chicago.
Device-independent color support in Chicago will allow you to ensure that your graphic colors are consistent across all different types of video cards and displays.

2 Easy to Support

Every support call might as well be a lost sale.

Providing customer support for games – or any software, for that matter – is very, very costly. In fact, the margin that game developers typically make from the sale of a game are generally low enough that the cost of one support call for installation or configuration can actually eliminate the profit earned from that sale.

Developing your game for Chicago and/or Microsoft Windows 3.1 will help you decrease support costs in four ways:

- Windows **memory management** busts the 640K conventional memory barrier for you, gives you up to gigabytes' worth of linear 32-bit addressable memory and eliminating the need to fiddle with customers' CONFIG.SYS file during setup. A huge portion of all support calls relate to setup.
- Windows **device independence** frees you from a lot of card-specific coding – and support. Use the standard Windows APIs and your product will work with all Windows-compatible devices.
- **Not your dime.** Importantly, if you've written your game for Windows and your user's sound card doesn't go "whoooosh!" when it ought to do so, your game isn't the culprit. Microsoft (or the sound card maker) gets the phone call, not you.
- In Chicago, a new feature called **Plug and Play** will make it easier for users to add devices – such as CD-ROM drives or sound cards – to their systems. This will help to further increase the market for games, facilitate upgrades, and take support pressure off of game developers.

Windows offers device-independent support for:

- CD-ROM drives
- Sound cards
- Video Displays
- Digital video acceleration boards (including MPEG)
- Printing
- Networking
- Modem
- Joystick, and
- Pointing devices including the mouse (of course)

3 Powerful Development Environment

The tools for writing Windows code have evolved greatly over the last few years. We aren't saying that your job is in danger − the code doesn't write itself. But some of the least rewarding parts have been substantially automated, and there are good tools now available to help you write code that you can reuse from one project to the next. For example:

- **Microsoft Visual C++.** Provides a fully integrated graphic development environment for Windows applications that makes it simple to create a GUI application utilizing sophisticated functionality such as networking, WinG, Object Linking and Embedding, sound, digital video, and so forth.
- **Object Linking and Embedding (OLE)** is an object technology made available for all Microsoft operations systems and the Mac that greatly facilitates the exchange of information and functionality between unrelated applications. The availability of this technology creates game possibilities previously unimaginable in the MS-DOS world, such as:
- Drag & Drop monsters from one game space to another;
- provide standard interfaces that others can use to develop or extend functionality for your game worlds;
- embedding game sessions into mail messages that can automatically connect you over a network or modem, etc.

4 Built-in Digital Video Support

For the past several years, Microsoft has been developing a high-performance architecture for digital video − Microsoft Video for Windows.

In the past, Microsoft Video for Windows has been sold separately (principally as a Software Developers' Kit). With the release of Chicago, it will be built right into the operating system. For the first time, the ability to play digital video will ship with every copy of Microsoft Windows.

If you include digital video in your game, Windows can play it back for your customers, regardless of the display board that the customer may have installed. Your customers don't need special hardware to play your game − any VGA card will do.

More information on Microsoft Video for Windows is available on MSDN and in the Microsoft Video for Windows software developers kit.

5 High Quality, Flexible Sound

Microsoft Windows offers device independent sound allowing applications to call standard APIs to drive sound boards without worrying about the hardware-specific details. The high-level MCI sound APIs make it relatively straightforward to play a given sound with minimal coding effort. The low-level WAV APIs provide more precise control over arbitrary digital sound. MIDI APIs are also provided.

For mixing sound, Microsoft offers a library called WAVEMIX.DLL, which can merge up to four channels of sound into a single stream on the fly. WAVEMIX can be found on CompuServe and on the Microsoft Windows Multimedia Jumpstart CD.

To make the burden of storing and playing sound less onerous, Chicago includes a family of sound compression technologies ("codecs"). These codecs can be divided into two groups:

- Music-oriented codecs (such as IMADPCM) are included that allow close to CD-quality sound to be compressed to about one-quarter size.

- Voice-oriented codecs (such as TrueSpeech) are included to allow very, very efficient compression of voice data.

This support for compressed sound is two-way − you can play sound from a compressed sound file, or you can compress a sound file (using the built-in sound recording and editing utility). If you have a microphone, you can turn on voice compression when recording so that your file is compressed in real-time.

6 Support for Multi-player Games

The much-touted information superhighway holds great promise in lots of areas, but one of the *least* controversial is the opportunity it might provide for cool multi-player games. Why wait for the information superhighway? Chicago offers two technologies that make Windows-based multi-player games viable right away:

- **Networks** offer great promise for multi-player games, but network support has been added to relatively few MS-DOS-based games today. This is principally for technical reasons − there are scores of different network hardware and software challenges, and even getting a game to run can be frustrating and expensive to support. It's much, much simpler in Windows − you don't have to write the networking code, you don't have to work around the network's memory space, and you don't have to take the network support calls. A Windows technology called WinSockets makes it possible to write games for a broad variety of network types including Novell, Banyan Vines, Windows for Workgroups, LAN Manager, TCP/IP and others without worrying about which one is in use.
- **Modems.** In addition to easy modem setup and configuration, Chicago provides support for a new modem technology called VoiceViewThis technology lends itself well to games that involve "taking turns," and will help to put the appeal of human interaction into modem-based games.

VoiceView will be shipped as a standard feature of virtually every modem in 1995

Until now, modem users have had no choice − they can either talk on the phone or use their modem, but not both. (If you want to talk, you generally have to disconnect the modem and call back, or else get a second phone line.) This makes modem-based 2-player games pretty unappealing, because talking to your opponent is half the fun, right?

With VoiceView, you can play *and* talk to your opponent in the same phone call. Here's how it works:

- Run one phone line from the wall to the "in" port of your VoiceView-capable modem, and another from the "out" port to your phone.
- Turn on your PC and call up your friend (who also has a VoiceView-capable modem).
- Talk for as long as you like. When you feel like it, launch a 2-player game − we'll use chess as an example.
- When you make a move, you will hear a brief beep, and your modems will take over the phone line for a moment. (You can't talk while the modems are conversing). A simple message like a chess move takes less than one second to communicate.
- When you're done (or have to go eat dinner), hang up like you always do.

7 Synchronization

Synchronization of sound with events is crucial for cutting-edge games. In Chicago, you can write 32-bit games that use threads to manage processes that occur simultaneously (such as sound, scoring and animation). In the past, many game developers have written their own multitasking engines under MS-DOS to build this sort of functionality. The multitasking support in Chicago will free you from this low-level coding so that you can invest more of your effort on features.

Chicago has a default preemptive thread scheduling grain of 20 ms, meaning that if there are not a lot of other background activities, a foreground application (or high-priority thread) can be assured of constant and frequent attention from the OS. If that isn't fast enough, it's possible to set the grain as fast as 1ms. Chicago provides fine event control and scheduling objects that facilitate writing tightly synchronized multitasking applications.

The ability to manage sound as a separate thread of your program allows multimedia titles and games to have a more smooth, finished feel to them. For example, a game might have a thread that plays background music continuously during game play. This would help smooth out the breaks between scenes, when the game is loading new data — on another thread of the program. Threading provides for asynchronous I/O that makes it possible to carry on sound and animation while hitting the network or file system at the same time.

Another area of synchronization that is important to a successful action game is synchronization of input with action. Again, using threading you can control the polling rate of input devices and ensure that your game feels crisp and responsive. In addition to a thread scheduler, Chicago also provides access to a configurable event timer that can generate clock messages at up to 1ms.

8 Toys

Support for game devices will be built right into Chicago. Aside from the obvious support for the mouse and the keyboard, Chicago will also include built-in support for joysticks.

9 3D for Windows

Windows NT (Daytona) will ship with the industry standard OpenGL libraries included as a standard part of the Windows Graphic API. We anticipate that this will make Windows NT a great authoring environment for 3D software. Microsoft will also make the OpenGL libraries available on Chicago, so that the same authoring tools can run on both Chicago and Windows NT. To enable the market for 3D accelerators, Microsoft will publish the 3D-DDI (device driver interface), which will make it possible for hardware vendors to accelerate 3D graphics for MS-DOS, Windows, and Windows NT. Our OpenGL implementation will utilize hardware acceleration via the 3D-DDI whenever it is available.

The 3D DDI is an open interface, enabling other 3D rendering APIs such as HOOPS and PeX to coexist with OpenGL on Windows. Although general-purpose 3D libraries are not practical for most games today, we hope that our support for OpenGL and the 3D DDI will rapidly grow the installed base of 3D hardware accelerators.

Appendix A: PRELIMINARY WinG Documentation

Why WinG?

Although business applications such as word processors and spreadsheets have moved overwhelmingly to Windows, MS-DOS remains the operating system of choice for games. These applications have not made the transition to Windows largely for performance reasons − in a word, *speed.* The performance of games under Windows has suffered because of restrictions placed on the programmer by GDI's device independence, by the windowed environment, and by the inability of general graphics libraries to provide the necessary speed.

Most MS-DOS game programmers use knowledge specific to their application and their hardware to write optimized graphics routines. Until now, Windows programmers could not use such methods because GDI prevents access to device-specific surfaces; programmers can not draw directly onto the surface of a GDI device context.

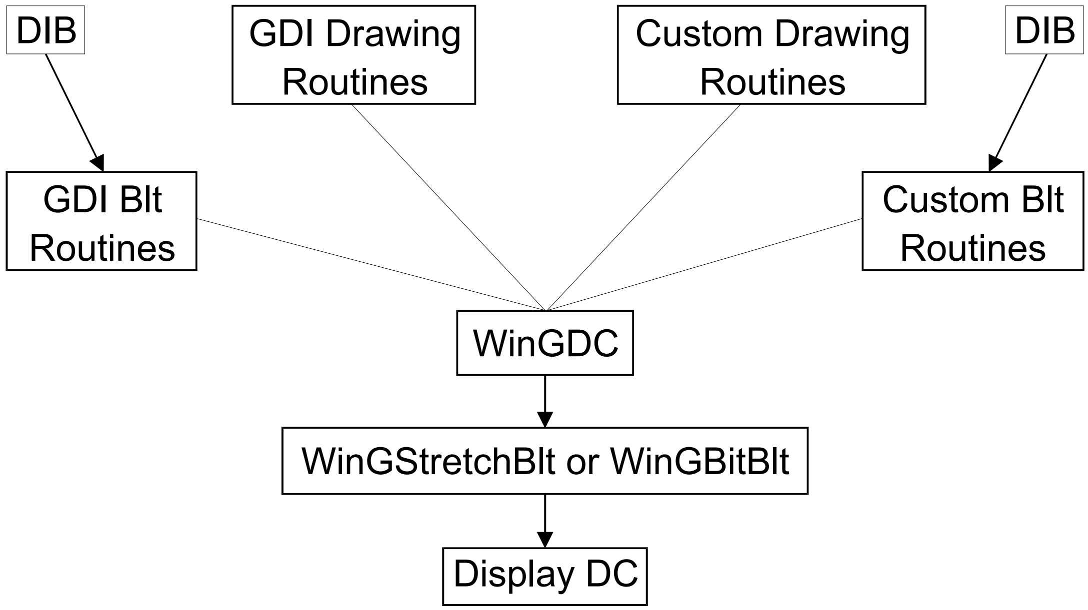
WinG (pronounced "Win Gee") is an optimized library designed to enable high-performance graphics techniques under Windows 3.1, Chicago, and Windows NT. WinG has been developed as a key component of the Microsoft Windows Multimedia Initiative.

WinG allows the programmer to create a GDI-compatible HBITMAP with a Device Independent Bitmap (DIB) as the drawing surface. Programmers can use GDI or their own code to draw onto this bitmap, then use WinG to transfer it quickly to the screen. WinG also provides halftoning APIs that use the standard Microsoft halftone palette to support simulation of true color on palette devices.

Off-screen Drawing With WinG

WinG introduces a new type of device context, the WinGDC, that can be used like any other device context. Unlike other DCs, programmers can retrieve a pointer directly to the WinGDC drawing surface, its BITMAPINFOHEADER, and its color table. They can also create and select new drawing surfaces for the WinGDC or modify the color table of an existing surface. DIBs become as easy to use as device-specific bitmaps and compatible DCs, and programmers can also draw into them using their own routines.

Most often, applications will use WinGCreateDC to create a single WinGDC and will use WinGCreateBitmap to create one or more WinGBitmaps into which they compose an image. The application will typically draw into this buffer using DIB copy operations, GDI calls, WinG calls, and custom drawing routines, as shown here.



A double-buffering architecture for WinG

Once DIB composition for the current frame is complete, the application will copy the WinGDC buffer to the screen using WinGStretchBlt or WinGBitBlt. This double-buffering architecture minimizes flicker and provides smooth screen updates.

Many games and animation applications draw their characters using sprites. On arcade machines, sprite operations are performed in hardware. Under DOS with a VGA, games simulate sprite hardware using transparent blts into an off-screen buffer. The DOGGIE sample application (in the SAMPLES\DOGGIE directory of the WinG development kit) uses WinG in the same way to perform transparent blts to a WinGDC and includes source code for an 8-bit to 8-bit TransparentDIBits procedure.

Using GDI With WinGDCs

WinG allows drawing onto the DIB surface of a WinGDC with GDI, but there are some anomalies to keep in mind.

1 Most importantly, GDI does NOT regard WinGDCs as palette devices. WinGDCs are actually RGB devices with a fixed 256-color color table. You happen to be able to modify the device color table using the WinGSetDIBColorTable API, but the color table is considered static by GDI. You can't select or realize palettes in a WinGDC. The Palette Animation With WinG article describes how to match a given palette to a WinGDC color table.

2 Drawing with GDI on a WinGDC surface does not always produce a pixel-perfect duplicate of the image you would see using GDI on a display device. The images will be similar, but some stray pixels will remain if you XOR the two images together.

Brushes realized in a WinGDC will be aligned to the upper left corner of the WinGDC whereas brushes used in screen DCs are aligned to the upper left corner of the screen. This means that when you blt a WinGDC that has been filled with a pattern into a screen DC that has been filled with the same pattern, the patterns will not necessarily align correctly.

If you have this problem, you can either change the brush origins and re-realize the brushes in either DC (see the section “1.6.8 Brush Alignment” in the Windows SDK Programmer’s Reference Volume 1, also available on the Microsoft Developer Network

CD) or you can make off-screen brushes align correctly with on-screen brushes by blting the WinGDC to a brush-aligned position on the screen. For example, an 8x8 brush pattern can be correctly aligned to the screen by blting the WinGDC to an x, y position when x and y are both multiples of 8.

Halftoning With WinG

WinG allows applications to simulate true 24-bit color on 8-bit devices through the WinG halftoning support APIs, WinGCreateHalftonePalette and WinGCreateHalftoneBrush.

The halftone palette is an identity palette containing a carefully selected ramp of colors optimized for dithering true color images to 8-bit devices. The WinGCreateHalftonePalette function returns a handle to a halftone palette which applications can select and realize into a display device context to take advantage of the halftoning capabilities offered by WinG.

The brushes returned by the WinGCreateHalftoneBrush API use patterns of colors in the halftone palette to create areas of simulated true color on 8-bit devices into which the halftone palette has been selected and realized. The CUBE sample application (in the SAMPLES\CUBE subdirectory of the WinG development kit) uses halftoned brushes to generate a wide range of shaded colors on an 8-bit display.

The halftone palette gives applications a framework for dithering 24-bit images to 8-bit devices. The palette itself is a slightly modified 2.6-bit-per-primary RGB cube, giving 216 halftoned values. The 256-color halftone palette also contains the twenty static colors and a range of gray values.

Given a 24-bit RGB color with 8 bits per primitive, you can find the index of the nearest color in the halftone palette using the following formula:

```
HalftoneIndex = (Red / 51) + (Green / 51) * 6 + (Blue / 51) * 36;
HalftoneColorIndex = aWinGHalftoneTranslation [HalftoneIndex];
```

The aWinGHalftoneTranslation vector can be found in the HALFTONE source code. The HALFTONE sample (in the SAMPLES\HALFTONE subdirectory of the WinG development kit) applies an ordered 8x8 dither to a 24-bit image, converting it to an 8-bit DIB using the WinG Halftone Palette.

Applications should avoid depending on a specific ordering of the halftone palette by using PALETTERGB instead of PALETTEINDEX to refer to entries in the palette.

Maximizing Performance With WinG

Here is the WinG Programmer's Guide to Achieving WinG Nirvana, the Top Ten ways to maximize blt performance under Windows using WinG.

10. Take Out Your Monochrome Debugging Card

Eight bit monochrome video cards can turn the 16 bit 8 MHz ISA bus into an 8 bit 4 MHz PC bus, cutting your video bus bandwidth by up to 75%. Monochrome cards are an invaluable aid when debugging graphical applications, but when timing or running final tests, remove the card for maximum speed.

9. Store WinGBitmap Surface Pointer and BITMAPINFO

WinGCreateBitmap takes a BITMAPINFO, creates an HBITMAP, and returns a pointer to the new bitmap surface. Store the BITMAPINFO and pointer at creation time with the HBITMAP rather than call WinGGetDIBPointer when you need it.

8. Don't Make Redundant GDI Calls

GDI objects such as brushes, fonts, and pens, take time to create, select, and destroy. Save time by creating frequently used objects once and caching them until they are no longer needed. Move the creation and selection of objects as far out of your inner loops as possible.

7. Special Purpose Code May Be Faster Than GDI

There may be many ways to accomplish a given graphics operation using GDI or custom graphics code in your application. Special purpose code can be faster than the general purpose GDI code, but custom code often incurs associated development and testing overhead. Determine if GDI can accomplish the operation and if the performance is acceptable for your problem. Weigh the alternatives carefully and see number 6 below.

6. Time Everything, Assume Nothing (Ruminations on good coding practices)

Software and its interactions with hardware are complex. Don't assume one technique is faster than another; time both. Within GDI, some APIs do more work than others, and there are sometimes multiple ways to do a single operation—not all techniques will be the same speed.

Remember the old software development adage: 90% of your time is spent executing 10% of the code. If you can find the 10% through profiling and optimize it, your application will be noticeably faster.

Timing results may depend on the runtime platform. An application's performance on your development machine may be significantly different from its performance on a different runtime machine. For absolute maximum speed, implement a variety of algorithms, time them at runtime or at installation, and choose code paths accordingly. If you choose to time at installation, remember that changes to video drivers and hardware configuration after your application has been installed can have a significant effect on runtime speed.

5. Don't Stretch

Stretching a WinGBitmap requires more work than just blting it. If you must stretch, stretching by factors of 2 will be fastest.

On the other hand, if your application is pixel-bound (it spends more time writing pixels to the bitmap than it does blting), it may be faster to stretch a small WinGBitmap to a larger window than it is to fill and blt a WinGBitmap with the same dimensions as the window. Your application can respond to the WM_GETMINMAXINFO message to restrict the size of your window if you don't want to deal with this problem.

4. Don't Blt

"The fastest code is the code that isn't called." Blt the smallest area possible as seldom as possible. Of course, figuring out the smallest area to blt might take longer than just blting a larger area. For example, a dirty rectangle sprite system could use complex algorithms to calculate the absolute minimum rectangles to update, but it might spend more time doing this than just blting the union of the dirty areas. The runtime environment can affect which method is faster. Again, time it to make sure.

3. Don't Clip

Selecting GDI clip regions into the destination DC or placing windows (like floating tool bars) over the destination DC can slow the blt speed.

Clip regions may seem like a good way to reduce the number of pixels actually sent to the screen, but someone has to do the work. As number 4 and number 7 discuss above, you may be better off doing the work yourself rather than using GDI.

An easy way to test your application's performance when clipped is to start the CLOCK.EXE program supplied with Windows. Set it to Always On Top and move it over your client area.

2. Use an Identity Palette

WinGBitmaps without identity palettes require a color translation per pixel when blted. 'Nuff said.

See the Using an Identity Palette article for specific information about what identity palettes are, how they work, and how you can use them.

1. Use the Recommended DIB Format

WinG adapts itself to the hardware available at runtime to achieve optimum performance on every platform. Every hardware and software combination can be different, and the best way to guarantee the best blt performance is to use the DIB parameters returned by WinGRecommendDibFormat in calls to WinGCreateBitmap. If you do this, remember that your code must support both bottom-up and top-down DIB formats. See the DIB Orientation article for more information on handling these formats.

DIB Orientation

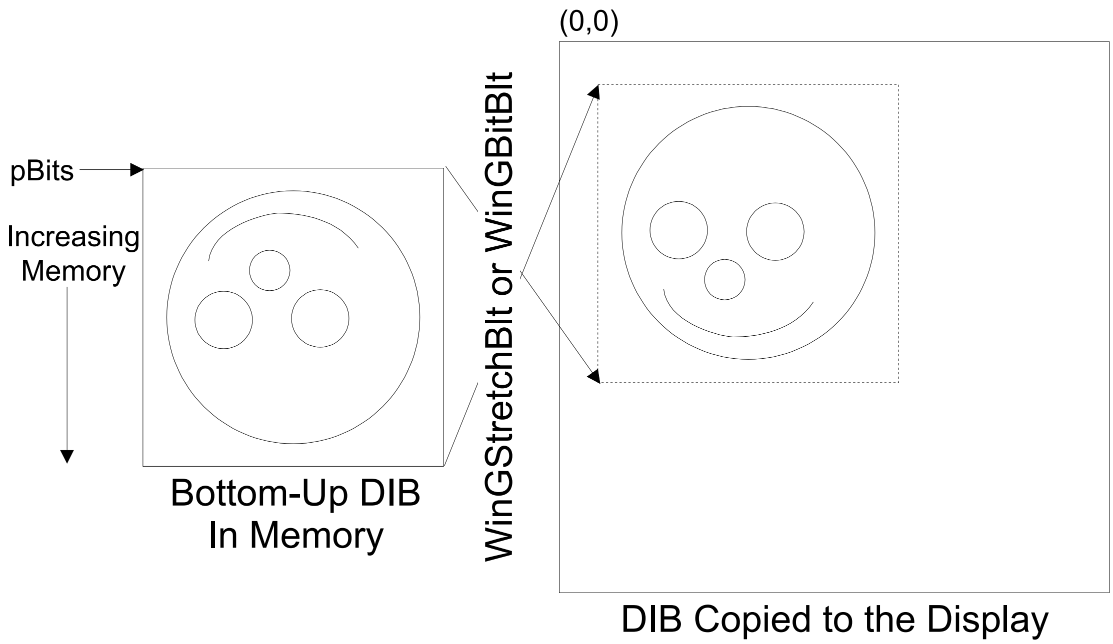
The most frustrating thing about working with DIBs is that DIBs are usually oriented with the bottommost scanline stored first in memory, the exact opposite of the usual device-dependent bitmap orientation. This standard type of Windows DIB is called a bottom-up DIB.

WinG hides the orientation of DIBs from an application unless the application wants to know. Drawing into a WinGDC using GDI functions and blting the WinGDC to the display using either of the WinG DIB copy commands (WinGStretchBlt or WinGBitBlt) results in an image that is almost identical to one created using GDI to draw directly onto a display DC. See the Using GDI With WinGDCs article for more information.

If you don't plan on writing custom drawing routines and will not be using existing Windows 3.1 DIB-to-screen functions (such as StretchDIBits or SetDIBitsToDevice), you can skip the rest of this section.

If you do plan on writing custom drawing routines or just want to know how they work, this section will begin to alleviate the confusion. The Microsoft Technical Articles "DIBs and Their Use" by Ron Gery and "Animation in Windows" by Herman Rodent will flesh out the ideas presented here, provide helpful advice, and describe DIBs in depth. The TRIQ sample code from Microsoft's GDI Technical Notes shows how to draw triangles and quads into a memory DIB. These articles are listed in the section Further Reading.

Confusion with bottom-up DIBs inevitably stems from the fact that the bottommost scanline is stored first in memory, giving a coordinate space where (0, 0) is the lower left corner of the image. Windows uses (0, 0) as the upper left corner of the display and of device dependent bitmaps, meaning that the Y coordinates of bottom-up DIBs are inverted. In the diagram below, the smiling face casts its gaze towards the DIB origin, but when translated to the display with WinGStretchBlt or WinGBitBlt, it looks away from the display origin.



Bottom-Up DIBs are flipped when copied to the display

WinGStretchBlt, WinGBitBlt, StretchDIBits, and SetDIBitsToDevice invert the bottom-up DIB as they draw it to the screen.

For an 8-bit bottom-up DIB, the address in memory from which the screen pixel (X, Y) is retrieved can be found with these equations:

```
// Calculate actual bits used per scan line
DibWidthBits = (UINT)lpBmiHeader->biWidth *
(UINT)lpBmiHeader->biBitCount);
// And align it to a 32 bit boundary
DibWidthBytes = ((DibWidthBits + 31) & (~31)) / 8;
pPixelXY = DibAddr + (DibHeight - 1 - Y) * DibWidthBytes + X
```

where DibAddr is the base address of the DIB, DibHeight is the height of the DIB, lpBmiHeader is a pointer to a BITMAPINFOHEADER describing the DIB, and DibWidthBytes is the DWORD-aligned offset of bytes in memory from any X in one scanline to any X in the next scanline.

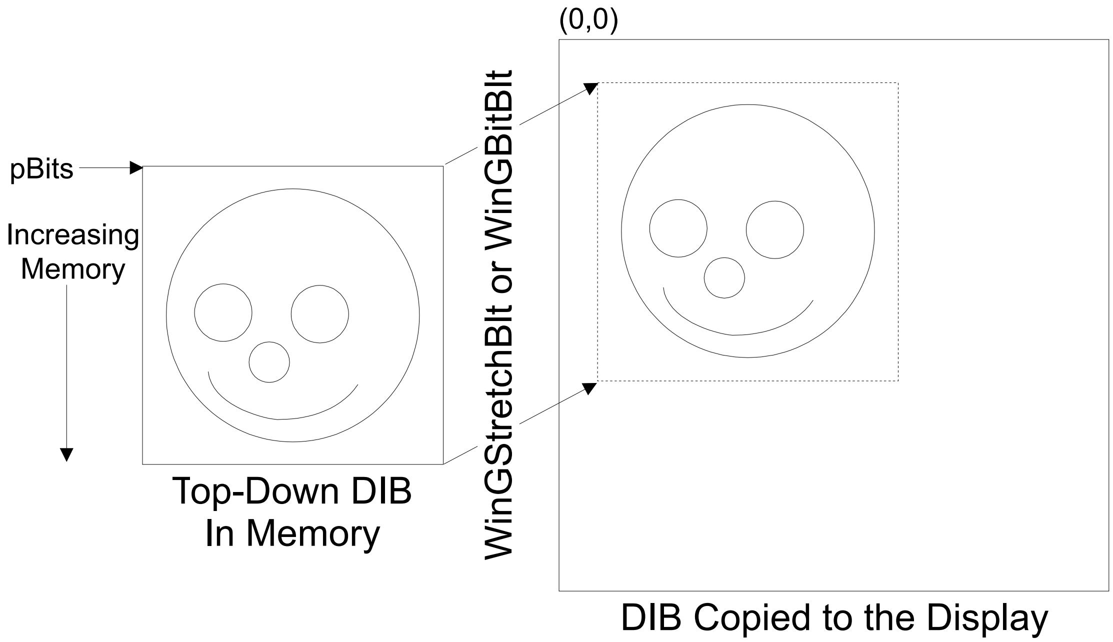
Top-Down DIBs

Another kind of DIB, called a top-down DIB, is stored with the same orientation as most device-dependent bitmaps: the first scanline in memory is the top of the image. Top-down DIBs are identified by a negative biHeight entry in their BITMAPINFOHEADER structures.

Sometimes, WinG can greatly improve the speed of a DIB-to-screen copy by using a top-down DIB because it can avoid inverting the DIB to a device-dependent format. When this is the case, WinGRecommendDIBFormat will return a negative value in the biHeight field of the passed BITMAPINFOHEADER structure.

If you are writing custom DIB drawing routines, you will have to handle top-down DIBs for best performance because there is a good chance that WinG will recommend them with WinGRecommendDibFormat.

WinGStretchBlt and WinGBitBlt recognize top-down DIBs and handle them correctly, but **Windows 3.1 functions such as StretchDIBits and SetDIBitsToDevice will not work properly with top-down DIBs unless you intentionally mirror the image.**



Top-Down DIBs are copied directly to the display

For an 8-bit top-down DIB, the memory address of the pixel (X, Y) can be found with this equation:

PixelAddr = DibAddr + Y * DibWidthBytes + X

where DibAddr is the base address of the DIB and DibWidthBytes is the DWORD-aligned offset of bytes in memory from the beginning of one scanline to the next.

The PALANIM sample application (in the SAMPLES\PALANIM subdirectory of the WinG development kit) includes a routine to draw horizontal lines into a DIB. To do this, it determines the orientation of the target DIB and performs its calculations accordingly.

The DOGGIE sample application (in the SAMPLES\DOGGIE subdirectory of the WinG development kit) includes a routine to copy one DIB into another with a transparent color. The assembly function that does this also behaves well with both DIB orientations.

Using an Identity Palette

The Windows Palette Manager, described in depth in Ron Gery's technical article "The Palette Manager: How and Why" (see the Further Reading section for details) arbitrates conflicts between Windows applications vying for color entries in a single hardware palette (known as the system palette). It gives each application it's own virtual 256-color palette, called a logical palette, and translates entries in the logical palette to entries in the system palette as they are needed for blting images to the screen.

An identity palette is a logical palette which exactly matches the current system palette. An identity palette does not require translation of palette entries, so using an identity palette can drastically improve the speed with which you can blt WinGDCs to the screen.

The WinG Halftone Palette is an identity palette. This article describes how to create your own identity palettes for maximum WinG blt speed.

Static Colors

The Palette Manager reserves a number of colors in the palette, called the static colors, which it uses to draw system elements such as window captions, menus, borders, and scroll bars. An identity palette must include the static colors in the appropriate palette entries.

The display driver defines the actual RGB values of the static colors, so they must always be determined at run time. The GetSystemPaletteEntries will retrieve the colors currently in the system palette, and you can isolate the static colors using the SIZEPALETTE and NUMCOLORS capability indices with GetDeviceCaps and a little knowledge of how the Palette Manager works.

The static colors are split in half and stored at either end of the system palette. If there are *nColors* possible entries in the system palette and there are *nStaticColors* static colors, then the static colors will be found in entries 0 through *nStaticColors*/2 - 1 and entries *nColors* - *nStaticColors*/2 through *nColors*-1 in the system palette. Typically, there are 20 static colors, found in indices 0-9 and 246-255 of a 256-color palette. The peFlags portion of these PALETTEENTRY structures must be set to zero.

The SetSystemPaletteUse API turns use of the static colors on and off for the system. Using SYSPAL_STATIC, 20 entries will be reserved in the palette. SYSPAL_NOSTATIC reserves only 2 entries, which must be mapped to black and white. See the Accessing a Full Palette Using SYSPAL_NOSTATIC article for more information.

Other Colors

The remaining non-static colors in the logical palette may be defined by the application, but they must be marked as PC_NOCOLLAPSE (see the PALETTEENTRY documentation for a description) to ensure an identity palette.

A palette containing the static colors in the appropriate entries with the remaining entries marked PC_NOCOLLAPSE, selected and realized into a DC, becomes an identity palette. Because no translation to the system palette is required, the Palette Manager can step aside gracefully and leave you to achieve maximum blt bandwidth.

Creating an Identity Palette

The CreateIdentityPalette() function below shows how to create an identity palette from a an array of RGBQUAD structures. Before you realize an identity palette for the first time, it may be a good idea to clear the system palette by realizing a completely black palette, as the ClearSystemPalette() function below does This will ensure that palette-managed applications executed before your application will not affect the identity mapping of your carefully constructed palette.

To make sure that you have successfully created and are using an identity palette, you can tell WinG to send debugging messages to the standard debug output, as described in the Debugging With WinG article.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) uses these routines to create a 256-entry identity palette filled with a wash of color.

Click Here to copy the CreateIdentityPalette() code sample to the clipboard.

Click Here to copy the ClearSystemPalette() code sample to the clipboard.

```
HPALETTE CreateIdentityPalette(RGBQUADWin31_API aRGB[], int nColors)
{
int i;
struct {
WORD Version;
WORD NumberOfEntries;
        PALETTEENTRY aEntries[256];
} Palette =
    {
```

```
0x300,
256
      };

//*** Just use the screen DC where we need it
      HDC hdc = GetDC(NULL);

//*** For SYSPAL_NOSTATIC, just copy the color table into
//*** a PALETTEENTRY array and replace the first and last entries
//*** with black and white
      if (GetSystemPaletteUse(hdc) == SYSPAL_NOSTATIC)
      {
//*** Fill in the palette with the given values, marking each
            //*** as PC_NOCOLLAPSE
for(i = 0; i < nColors; i++)
            {
Palette.aEntries[i].peRed = aRGB[i].rgbRed;
Palette.aEntries[i].peGreen = aRGB[i].rgbGreen;
Palette.aEntries[i].peBlue = aRGB[i].rgbBlue;
Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
            }

//*** Mark any unused entries PC_NOCOLLAPSE
for (; i < 256; ++i)
            {
Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
            }

//*** Make sure the last entry is white
//*** This may replace an entry in the array!
Palette.aEntries[255].peRed = 255;
Palette.aEntries[255].peGreen = 255;
Palette.aEntries[255].peBlue = 255;
Palette.aEntries[255].peFlags = 0;

//*** And the first is black
//*** This may replace an entry in the array!
Palette.aEntries[0].peRed = 0;
Palette.aEntries[0].peGreen = 0;
Palette.aEntries[0].peBlue = 0;
Palette.aEntries[0].peFlags = 0;
      }
else
//*** For SYSPAL_STATIC, get the twenty static colors into
//*** the array, then fill in the empty spaces with the
//*** given color table
      {
int nStaticColors;
int nUsableColors;

            //*** Get the static colors from the system palette
            nStaticColors = GetDeviceCaps(hdc, NUMCOLORS);
            GetSystemPaletteEntries(hdc, 0, 256, Palette.aEntries);

//*** Set the peFlags of the lower static colors to zero
nStaticColors = nStaticColors / 2;
for (i=0; i<nStaticColors; i++)
```

```
Palette.aEntries[i].peFlags = 0;
//*** Fill in the entries from the given color table
nUsableColors = nColors - nStaticColors;
for (; i<nUsableColors; i++)
        {
Palette.aEntries[i].peRed = aRGB[i].rgbRed;
Palette.aEntries[i].peGreen = aRGB[i].rgbGreen;
Palette.aEntries[i].peBlue = aRGB[i].rgbBlue;
Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
        }

//*** Mark any empty entries as PC_NOCOLLAPSE
for (; i<256 - nStaticColors; i++)
Palette.aEntries[i].peFlags = PC_NOCOLLAPSE;
//*** Set the peFlags of the upper static colors to zero
for (i = 256 - nStaticColors; i<256; i++)
Palette.aEntries[i].peFlags = 0;
    }

//*** Remember to release the DC!
    ReleaseDC(NULL, hdc);

//*** Return the palette
    return CreatePalette((LOGPALETTE *)&Palette);
}


void ClearSystemPalette(void)
{
//*** A dummy palette setup
struct
    {
WORD Version;
WORD NumberOfEntries;
        PALETTEENTRY aEntries[256];
} Palette =
    {
0x300,
256
    };

    HPALETTE ScreenPalette = 0;
    HDC ScreenDC;
int Counter;
//*** Reset everything in the system palette to black
for(Counter = 0; Counter < 256; Counter++)
    {
Palette.aEntries[Counter].peRed = 0;
Palette.aEntries[Counter].peGreen = 0;
Palette.aEntries[Counter].peBlue = 0;
Palette.aEntries[Counter].peFlags = PC_NOCOLLAPSE;
    }

//*** Create, select, realize, deselect, and delete the palette
    ScreenDC = GetDC(NULL);
    ScreenPalette = CreatePalette((LOGPALETTE *)&Palette);
ScreenPalette = SelectPalette(ScreenDC,ScreenPalette,FALSE);
```

```
    RealizePalette(ScreenDC);
ScreenPalette = SelectPalette(ScreenDC,ScreenPalette,FALSE);
    DeleteObject(ScreenPalette);
    ReleaseDC(NULL, ScreenDC);
}
```

Palette Animation With WinG

Palette animation creates the appearance of motion in an image by modifying entries the system palette, resulting in color changes in the displayed image. Carefully arranged and animated palette entries can produce motion effects such as running water, flowing lava, or even motion of an object across the screen.

The AnimatePalette function in Windows replaces entries in a logical palette. When the modified palette is realized, the colors in the palette will be remapped, and colors on the display will change.

Because every DIB and WinGBitmap has an associated color table which is translated to the system palette when the image is copied to the screen, DIBs blted after the palette is animated will not appear animated because their colors are translated to the new palette.

The Using an Identity Palette article discusses the creation of an identity palette which removes the need for color translation when blting. If a palette animating application went through the trouble to create the identity palette, it should maintain the identity mapping between the palette and the WinGDC by matching the WinGBitmap color table to the animated palette before blting. To do this, use WinGSetDibColorTable to keep the WinGBitmap color table synchronized with changes in the system palette.

Remember that any entries in a palette which are to be animated must be marked with the PC_RESERVED flag. This includes the PC_NOCOLLAPSE flag, so these entries can be included in an identity palette.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) performs a simple palette animation with an identity palette, making sure to update the WinGDC color table to match the palette before it blts using the following code, which copies the current logical palette (hpalApp) into the color table of the WinGDC (hdcOffscreen). Of course, if you create the palette yourself from an array of colors, there will be no need to call GetPaletteEntries because you could update the color table from the array you already have in memory. Also, in a palette animation that does not animate the complete palette, an application would not need to modify the entire palette and color table, as this code snippet does:

```
int i;
PALETTEENTRY aPalette[256];
RGBQUAD aPaletteRGB[256];

//*** BEFORE BLTING, match the DIB color table to the
//*** current palette to match the animated palette
GetPaletteEntries(hpalApp, 0, 256, aPalette);
//*** Alas, palette entries are r-g-b, rgbquads are b-g-r
for (i=0; i<256; ++i)
{
aPaletteRGB[i].rgbRed = aPalette[i].peRed;
aPaletteRGB[i].rgbGreen = aPalette[i].peGreen;
aPaletteRGB[i].rgbBlue = aPalette[i].peBlue;
aPaletteRGB[i].rgbReserved = 0;
}
WinGSetDIBColorTable(hdcOffscreen, 0, 256, aPaletteRGB);
```

Accessing a Full Palette Using SYSPAL_NOSTATIC

The Palette Manager usually reserves twenty static colors in the palette for use in drawing captions, menus, text, scroll bars, window frames, and other system elements. These static colors ensure a common color scheme across all applications, but this leaves only 236 palette entries available to each application. A Windows graphics application requiring a full palette of 256 colors has two options, outlined here.

The first option is to incorporate the static colors into the palette at runtime, knowing that the RGB values of the colors may change slightly from display driver to display driver. This means that the palette will vary slightly when the application runs on different platforms, but it ensures the consistent look and feel between the application and coexisting applications in the system.

The static colors are defined as follows:

Index	Color	Index	Color
0	Black	246	Cream
1	Dark Red	247	Light Gray
2	Dark Green	248	Medium Gray
3	Dark Yellow	249	Red
4	Dark Blue	250	Green
5	Dark Magenta	251	Yellow
6	Dark Cyan	252	Blue
7	Light Gray	253	Magenta
8	Money Green	254	Cyan
9	Sky Blue	255	White

If you can accept the limitation of including these colors in your palette and determining their exact RGB values at runtime (using GetSystemPaletteEntries), you can skip the rest of this article.

The second option is to tell the Window Manager to make 18 of the twenty static colors available to the application, with entry 0 remaining black and entry 255 remaining white. However, choosing to control those palette entries means you'll have some more intimate relations with the Palette Manager.

To change the use of the static colors in the system palette, you use the SetSystemPaletteUse API, passing either SYSPAL_STATIC or SYSPAL_NOSTATIC. Setting the palette use to SYSPAL_NOSTATIC gives you access to palette entries 1 through 254. Your palette must map entry 0 to RGB(0, 0, 0) and entry 255 to RGB(255, 255, 255), but black and white are standard in most palettes anyway.

Ordinarily, Windows uses entries 0-9 and 246-255 to draw captions, borders, menus, and text, and it will continue to do so after you've changed the RGB values of those palette entries unless you tell it to do otherwise. If you do not inform the operating system of your changes, your application and all others in the system will become very messy and your application will be condemned by its peers as unfriendly.

You want your application to be friendly to the operating system and to the other active applications. You can handle this in two ways: you can make your application a full-screen window with no controls, thereby taking over the entire screen and the full palette, or you can tell the operating system to use different palette entries to draw its captions, borders, menus, and text so that other visible windows do not appear completely strange. In either case, you must restore the static colors when your application becomes inactive or exits.

The following procedure handles the switch between SYSPAL_STATIC and SYSPAL_NOSTATIC for you, managing the mapping and remapping of the system colors for you through the Windows functions GetSysColor and SetSysColors. It stores the current mapping of the system colors before switching to SYSPAL_NOSTATIC mode and restores them after switching back to SYSPAL_STATIC mode.

To use the AppActivate() function in an application, call AppActivate((BOOL)wParam) in response to a WM_ACTIVATEAPP message and call

AppActivate(FALSE) before exiting to restore the system colors. This will set the system palette use and remap the system colors when your application is activated or deactivated.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) uses this function to take over the static colors at run time and clean up before it exits.

```
#define NumSysColors (sizeof(SysPalIndex)/sizeof(SysPalIndex[1]))
#define rgbBlack RGB(0,0,0)
#define rgbWhite RGB(255,255,255)

//*** These are the GetSysColor display element identifiers
static int SysPalIndex[] = {
    COLOR_ACTIVEBORDER,
    COLOR_ACTIVECAPTION,
    COLOR_APPWORKSPACE,
    COLOR_BACKGROUND,
    COLOR_BTNFACE,
    COLOR_BTNSHADOW,
    COLOR_BTNTEXT,
    COLOR_CAPTIONTEXT,
    COLOR_GRAYTEXT,
    COLOR_HIGHLIGHT,
    COLOR_HIGHLIGHTTEXT,
    COLOR_INACTIVEBORDER,
    COLOR_INACTIVECAPTION,
    COLOR_MENU,
    COLOR_MENUTEXT,
    COLOR_SCROLLBAR,
    COLOR_WINDOW,
    COLOR_WINDOWFRAME,
    COLOR_WINDOWTEXT
};

//*** This array translates the display elements to black and white
static COLORREF MonoColors[] = {
rgbBlack,
rgbWhite,
rgbWhite,
rgbWhite,
rgbWhite,
rgbBlack,
rgbBlack,
rgbBlack,
rgbBlack,
rgbBlack,
rgbWhite,
rgbWhite,
rgbWhite,
rgbWhite,
rgbBlack,
rgbWhite,
rgbWhite,
rgbBlack,
rgbBlack
};
```

```
//*** This array holds the old color mapping so we can restore them
static COLORREF OldColors[NumSysColors];

//*** AppActivate sets the system palette use and
//*** remaps the system colors accordingly.
void AppActivate(BOOL fActive)
{
    HDC hdc;
int i;
//*** Just use the screen DC
    hdc = GetDC(NULL);

//*** If the app is activating, save the current color mapping
//*** and switch to SYSPAL_NOSTATIC
    if (fActive && GetSystemPaletteUse(hdc) == SYSPAL_STATIC)
    {
//*** Store the current mapping
for (i=0; i<NumSysColors; i++)
OldColors[i] = GetSysColor(SysPalIndex[i]);
//*** Switch to SYSPAL_NOSTATIC and remap the colors
        SetSystemPaletteUse(hdc, SYSPAL_NOSTATIC);
        SetSysColors(NumSysColors, SysPalIndex, MonoColors);
    }
    else if (!fActive && GetSystemPaletteUse(hdc) == SYSPAL_NOSTATIC)
    {
//*** Switch back to SYSPAL_STATIC and the old mapping
        SetSystemPaletteUse(hdc, SYSPAL_STATIC);
        SetSysColors(NumSysColors, SysPalIndex, OldColors);
    }

//*** Be sure to release the DC!
    ReleaseDC(NULL,hdc);
}
```

WinGBitBlt

Copies an area from a specified device context to a destination device context. **WinGBitBlt** is optimized for copying WinGDCs to display DCs.

BOOL WinGBitBlt(HDC hdcDest, int nXOriginDest, int nYOriginDest, int nWidthDest, int nHeightDest, HDC hdcSrc, int nXOriginSrc, int nYOriginSrc)

Parameters

hdcDest	Identifies the destination device context.
nXOriginDest	X coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nYOriginDest	Y coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nWidthDest	Width of the source and destination rectangle..
nHeightDest	Height of the source and destination rectangle..
hdcSrc	Identifies the source device context.
nXOriginSrc	X coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.
nYOriginSrc	Y coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.

Return Value

The return value is non-zero if the function is successful. Otherwise, it is zero.

Comments

WinGBitBlt requires both DCs to use MM_TEXT mapping mode at the time of the call or the results may be unpredictable. At other times, any mapping mode may be used in either DC.

Maximizing Performance

You will get the highest performance from **WinGBitBlt** if you select a WinGBitmap created from header information supplied by a call to WinGRecommendDIBFormat.

WinGBitBlt is optimized for copying WinGDCs to the screen.

Clipping can slow **WinGBitBlt** down. In general, don't select clipping regions into or blt outside the boundaries of the source or destination DCs and avoid blting to an overlapped window if possible.

See Also

WinGStretchBlt WinGCreateDC WinGCreateBitmap WinGRecommendDIBFormat Maximizing Performance With WinG

WinGCreateBitmap

Creates a WinGBitmap for the given WinGDC using the specified header information.

HBITMAP WinGCreateBitmap(HDC *hWinGDC*, BITMAPINFO far **pHeader*, void far *far **ppBits*)

Parameters

hWinGDC	Identifies the WinG device context.
pHeader	Points to a BITMAPINFO structure specifying the width, height, and color table for the new WinGBitmap.
ppBitsr	If not 0, points to a pointer to receive the address of the new WinGDC DIB surface.

Return Value

Returns a handle to the new WinGBitmap DIB surface or 0 if it is unsuccessful.

Comments

Under Windows 3.1, **WinGCreateBitmap** will only create 8-bit-per-pixel surfaces.

If ppBits is 0, the address of the newly created bitmap will not be returned. WinGGetDIBPointer will also return this information.

pHeader must point to enough memory to hold a BITMAPINFOHEADER and a complete color table of RGBQUAD entries. The biClrUsed field of the BITMAPINFOHEADER specifies the number of colors in the color table; if it is zero, the maximum number of colors according to biBitCount are used if biBitCount is less than 24. For example, if biBitCount is 8 and biClrUsed is 0, 256 palette entries are expected. See the BITMAPINFOHEADER description in the Windows 3.1 SDK Reference for more information.

When an application has finished using a WinGBitmap, it should select the bitmap out of its WinGDC and remove the bitmap by calling DeleteObject.

The pointer to the WinGBitmap DIB surface returned by **WinGCreateBitmap** must not be freed by the caller. The allocated memory will be freed by a call to DeleteObject.

WinGCreateBitmap uses *pHeader* and the subsequent color table to create the drawing surface. WinG ignores the biClrImportant, biXPelsPerMeter, biYPelsPerMeter, and biSizeImage fields. WinG expects biCompression to be BI_RGB.

If the biHeight field of the passed BITMAPINFOHEADER is negative, **WinGCreateBitmap** will create a top-down DIB as the bitmap surface. See the article on DIB Orientation for a discussion of top-down and bottom-up DIBs.

An HBITMAP can only be selected into one device context at a time, and a device context can only have a single HBITMAP selected in at a time.

Maximizing Performance

To create a WinGBitmap that will maximize WinGBitBlt performance, use WinGRecommendDIBFormat to fill in the entries of *pHeader* before calling **WinGCreateBitmap**, remembering to modify the height and width to suit your needs.

Larger WinGBitmaps take longer to blt to the screen. Also, if the screen DC is clipped, for example by an overlapping window or by a selected clip region, the WinGDC will take longer to blt to the screen.

Using an identity palette that exactly matches the WinGBitmap's color table will greatly increase performance.

Example

The following code fragment shows how an application could create a WinGDC with an optimal 100x100 WinGBitmap selected for drawing, then delete it when it is no longer needed. Note that the WinGBitmap will initially have garbage in its color table—be sure to call WinGSetDIBColorTable before using the WinGDC.

The PALANIM sample (in the SAMPLES\PALANIM subdirectory of the WinG development kit) uses these routines, modified to create a 256x256 WinGDC, to allocate and free its drawing buffer.

```
HBITMAP ghBitmapMonochrome = 0;

HDC Create100x100WinGDC(void)
{
    HDC hWinGDC;
    HBITMAP hBitmapNew;
struct {
        BITMAPINFOHEADER InfoHeader;
        RGBQUAD ColorTable[256];
} Info;
void far *pSurfaceBits;

// Set up an optimal bitmap
    if (WinGRecommendDibFormat((BITMAPINFO far *)&Info) == FALSE)
return 0;
// Set the width and height of the DIB but preserve the
// sign of biHeight in case top-down DIBs are faster
Info.InfoHeader.biHeight *= 100;
Info.InfoHeader.biWidth = 100;
// Create a WinGDC and Bitmap, then select away
    hWinGDC = WinGCreateDC();
if (hWinGDC)
    {
hBitmapNew = WinGCreateBitmap(hWinGDC,
            (BITMAPINFO far *)&Info, &pSurfaceBits);
if (hBitmapNew)
        {
            ghBitmapMonochrome = (HBITMAP)SelectObject(hWinGDC,
hBitmapNew);
        }
else
        {
```

```
            DeleteDC(hWinGDC);
hWinGDC = 0;
        }
    }

return hWinGDC;
}

void Destroy100x100WinGDC(HDC hWinGDC)
{
    HBITMAP hBitmapOld;

if (hWinGDC && ghBitmapMonochrome)
    {
// Select the stock 1x1 monochrome bitmap back in
        hBitmapOld = (HBITMAP)SelectObject(hWinGDC,
ghBitmapMonochrome);
        DeleteObject(hBitmapOld);
        DeleteDC(hWinGDC);
    }
}
```

See Also

WinGCreateDC WinGRecommendDIBFormat CreateBitmapCreateCompatibleBitmap BITMAPINFO BITMAPINFOHEADER WinGGetDIBPointer CreateDIBSection Code Samples Off-screen Drawing With WinG Maximizing Performance With WinG

WinGCreateDC

Creates a WinG device context with the stock 1x1 monochrome bitmap selected.

    HDC WinGCreateDC(void)

Return Value

Returns the handle to a new WinGDC if successful. Otherwise, **WinGCreateDC** returns 0.

Comments

Device contexts created using **WinGCreateDC** must be deleted using the DeleteDC function. All objects selected into the WinGDC after it was created should be selected out and replaced with the original objects before the device context is deleted.

When a WinGDC is created, WinG automatically selects the stock 1x1 monochrome bitmap as its drawing surface. To begin drawing on the WinGDC, select a WinGBitmap created by the WinGCreateBitmap function into the WinGDC.

Maximizing Performance

WinGCreateDC has a fairly high overhead and is usually used to create a single off-screen DC. In general, programs will call **WinGCreateDC** once at startup then select new WinGBitmaps on WM_SIZE messages to the double-buffered window. Applications can use the WM_GETMINMAXINFO message to restrict the size of their window if necessary.

Compose frames into WinGDCs, then use WinGStretchBlt or WinGBitBlt to copy the WinGDC to the screen.

Example

See the WinGCreateBitmap API for sample code that uses **WinGCreateDC**.

See Also

WinGCreateBitmap CreateDC DeleteDC WM_SIZE WM_GETMINMAXINFO WinGStretchBlt WinGBitBlt CreateDIBSection Off-screen Drawing With WinG Maximizing Performance With WinG Code Samples

---

WinGCreateHalftoneBrush

Creates a dithered pattern brush based on the WinG halftone palette.

HBRUSH WinGCreateHalftoneBrush(HDC *hdc*, COLORREF *Color*, enum WING_DITHER_TYPE *DitherType*)

Parameters

hdc	Specifies the DC with which the brush should be compatible.
Color	Specifies the color to be approximated by the brush.
DitherType	Specifies the dither pattern for the brush. Can be one of: WING_DISPERSED_4x4 WING_DISPERSED_8x8 WING_CLUSTERED_4x4

Return Value

Returns a handle to a GDI brush if successful. Otherwise, **WinGCreateHalftoneBrush** returns 0.

Comments

This API is intended for simulating true color on 8-bit devices. It will create a patterned brush using colors from the halftone palette regardless of the color resolution of the target device. If *hdc* refers to a 24-bit device, **WinGCreateHalftoneBrush** will not return a solid brush of the given color, it will return a colored dither pattern using colors that appear in the halftone palette. On true-color devices, creating a solid brush that exactly matches the desired color is simple; **WinGCreateHalftoneBrush** lets you use the halftone patterns instead if you so desire.

A halftone brush approximates the requested *Color* using combinations of colors in the halftone palette. Larger dither patterns give a better approximation of the desired color but require more area to show the approximation. Quality is subjective, so programmers should experiment with different dither types to find the one that suits their needs.

If the target DC is a palette device and the WinG halftone palette has not been selected and realized into the target DC when a halftone brush is used, the visual results will be unpredictable. Use the WinGCreateHalftonePalette function to create a copy of the halftone palette, then select and realize it before using a halftone brush on a palette device.

The DISPERSED_nxn dither types create nxn patterns that approximate *Color* with a dispersed dot ordered dither.

The CLUSTERED_4x4 dither type creates a 4x4 pattern that approximates *Color* with a clustered dot ordered dither.

Always free GDI objects such as brushes by calling DeleteObject when the object is no longer needed.

Maximizing Performance

Avoid redundant creation, selection, and deletion of identical brushes as much as possible. If an application will be using the same brush repeatedly, it should create the brush once and save it for later use, deleting it when the application is complete.

Example

The CUBE sample application (in the SAMPLES\CUBE directory of the WinG Development Kit) allows the user to select the dither type for creating shaded brushes and provides a good experiment in using the different dither types.

See Also

WinGCreateHalftonePalette WING_DITHER_TYPE CreateDIBPatternBrush CreateSolidBrush Halftoning With WinG Using GDI With WinGDCs Code Samples

WinGCreateHalftonePalette

Creates an 8-bit palette used for halftoning images.
HPALETTE WinGCreateHalftonePalette(void)

Return Value

Returns the handle of a logical palettel containing the colors of the WinG halftone palette palette if successful. Otherwise, **WinGCreateHalftonePalette** returns 0.

Comments

The halftone palette should be selected into any DC into which the application will use WinG to halftone.

The WinG halftone palette is an identity palette: the logical palette indices and physical device indices are the same.

The halftone palette inverts correctly, so bitwise XORs invert colors properly.

See the Using an Identity Palette article for a discussion of identity palettes.

Maximizing Performance

Call **WinGCreateHalftonePalette** once at the beginning of your application. Select and realize the palette on WM_QUERYNEWPALETTE, WM_PALETTECHANGED, and WM_PAINT messages.

Example

The HALFTONE sample application (in the SAMPLES\CUBE directory of the WinG Development Kit) uses the halftone palette to dither 24-bit images to 8-bits using an 8x8 ordered dither.

See Also

WinGCreateHalftoneBrush WinGStretchBlt WinGBitBlt RealizePalette WM_QUERYNEWPALETTE WM_PALETTECHANGED Halftoning With WinG Using an Identity Palette Code Samples

WinGGetDIBColorTable

Returns the color table of the WinGBitmap currently selected into a WinGDC.
UINT WinGGetDIBColorTable(HDC *hWinGDC*, UINT *StartIndex*, UINT *NumberOfEntries*, RGBQUAD far **pColors*)

Parameters

hWinGDC	Identifies the WinG device context whose color table should be retrieved.
StartIndex	Indicates the first palette entry to be retrieved.
NumberOfEntries	Indicates the number of palette entries to retrieve.
pColors	Points to a buffer which receives the requested color table entries.

Return Value

Returns the number of palette entries copied into the given buffer or 0 if it failed.

Comments

The *pColors* buffer must be at least large enough to hold *NumberOfEntries* RGBQUAD structures.

Note that *StartIndex* indicates an entry in a palette array, which is zero-based. *NumberOfEntries* indicates a count, which is one-based. If *NumberOfEntries* is zero, no color table entries will be retrieved.

WinGGetDIBColorTable will return 0 for WinGBitmaps with more than 8 bits per pixel.

See Also

WinGSetDIBColorTable WinGCreateBitmap

WinGGetDIBPointer

Retrieves information about a WinGBitmap and returns a pointer to its surface.

void far *WinGGetDIBPointer(HBITMAP *hWinGBitmap*, BITMAPINFO far **pHeader*)

Parameters

hWinGBitmap	Identifies the WinGBitmap whose surface should be retrieved.
pHeader	If not 0, points to a buffer to receive the attributes and color table of the WinGDC.

Return Value

Returns a pointer to the bits of a WinGBitmap drawing surface if possible. Otherwise, **WinGGetDIBPointer** returns 0.

Comments

If it is supplied, *pHeader* must be large enough to hold a BITMAPINFOHEADER and enough RGBQUAD structures to hold the color table of the specified WinGBitmap.

If hWinGBitmap is not a WinGBitmap handle, this function will return 0 and *pHeader will remain unchanged.

Maximizing Performance

WinGCreateBitmap uses or returns the information returned by **WinGGetDIBPointer** as part of the creation process. If possible, applications should store the data when the WinGBitmap is created rather than calling **WinGGetDIBPointer** every time the information is required.

The address of a WinGBitmap surface will remain the same for the life of the WinGBitmap.

See Also

WinGCreateDC WinGCreateBitmap BITMAPINFO BITMAPINFOHEADER

WinGRecommendDIBFormat

Fills in the entries of a BITMAPINFO structure with values that will give maximum performance for memory-to-screen blts using WinG.

BOOL WinGRecommendDIBFormat(BITMAPINFO far **pHeader*)

Parameters

pHeader	Points to a BITMAPINFO structure to receive the recommended DIB format.

Return Value

Returns non-zero if successful. Otherwise, returns zero.

Comments

pHeader must point to enough memory to hold a BITMAPINFOHEADER. **WinGRecommendDIBFormat** will not return a color table.

For any combination of hardware and software, there will be one DIB format that WinG can copy fastest from memory to the screen. **WinGRecommendDibFormat** returns this optimal format, most important the recommended pixel format.

In many cases, WinG will find that it can copy a DIB to the screen faster if the DIB is in top-down format rather than the usual bottom-up format. **WinGRecommendDibFormat** will set the biHeight entry of the BITMAPINFOHEADER structure to -1 if this is the case, otherwise biHeight will be set to 1. See the DIB Orientation article for more information about these special DIBs.

WinGRecommendDIBFormat always recommends an 8-bit-per-pixel format under Windows 3.1. Other pixel formats are supported for Chicago and Windows NT. Code that uses this API should never assume that it will recommend an 8-bit format, as this may change depending on the run-time platform.

Example

See the WinGCreateBitmap API for sample code that uses **WinGRecommendDibFormat**.

See Also

WinGCreateBitmap BITMAPINFO BITMAPINFOHEADER Code Samples

WinGSetDIBColorTable

Modifies the color table of the currently selected WinGBitmap in a WinGDC.

UINT WinGSetDIBColorTable(HDC *hWinGDC*, UINT *StartIndex*, UINT *NumberOfEntries*, RGBQUAD far **pColors*)

Parameters

hWinGDC	Identifies the WinG device context whose color table should be modified.
StartIndex	Indicates the first palette entry to be changed.
NumberOfEntries	Indicates the number of palette entries to change.
pColors	Points to a buffer which contains the new color table values.

Return Value

Returns the number of palette entries modified in the specified device context or 0 if it failed.

Comments

The *pColors* buffer must hold at least *NumberOfEntries* RGBQUAD structures.

If you want to update the display immediately (for example, in palette animation), use AnimatePalette to modify the system palette and then call **WinGSetDIBColorTable** to match it or the WinGDC will be remapped when it is blted. See the Palette Animation With WinG article for more information and sample code that does this.

Note that *StartIndex* indicates an entry in a palette array, which is zero-based. *NumberOfEntries* indicates a count, which is one-based. If *NumberOfEntries* is zero, no color table entries will be modified.

Maximizing Performance

It is not necessary to call **WinGSetDIBColorTable** every time you call AnimatePalette. Only call this API if you are about to blt and the destination palette has changed since the last call to **WinGSetDIBColorTable**.

Example

See the section titled Palette Animation With WinG for sample code and discussion of using **WinGSetDIBColorTable** to perform palette animation.

The PALANIM sample, in the SAMPLES\PALANIM subdirectory of the WinG Development Kit, performs simple palette animation and maintains an identity palette throughout.

See Also

WinGGetDIBColorTable WinGCreateBitmap Palette Animation With WinG

WinGStretchBlt

Copies the source DC to the destination DC, resizing if necessary to fill the destination rectangle. Optimized for blting WinGDCs to screen DCs.

BOOL WinGStretchBlt(HDC hdcDest, int nXOriginDest, int nYOriginDest, int nWidthDest, int nHeightDest, HDC hdcSrc, int nXOriginSrc, int nYOriginSrc, int nWidthSrc, int nHeightSrc)

Parameters

hdcDest	Identifies the destination device context.
nXOriginDest	X coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nYOriginDest	Y coordinate of the upper-left corner of the destination rectangle in MM_TEXT client coordinates.
nWidthDest	Width of the destination rectangle..
nHeightDest	Height of the destination rectangle..
hdcSrc	Identifies the source device context.
nXOriginSrc	X coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.
nYOriginSrc	Y coordinate of the upper-left corner of the source rectangle in MM_TEXT client coordinates.
nWidthSrc	Width of the source rectangle.
nHeightSrc	Height of the source rectangle.

Return Value

Returns non-zero if successful, otherwise returns zero.

Comments

WinGStretchBlt requires both DCs to use MM_TEXT mapping mode at the time of the call or the results may be unpredictable. At other times, any mapping mode may be used in either DC.

WinGStretchBlt uses the STRETCH_DELETESCANS mode when expanding or shrinking an image, so stretched images may appear chunky.

Maximizing Performance

You will get the highest performance from **WinGStretchBlt** if you use a WinGBitmap created from header information supplied by a call to WinGRecommendDIBFormat.

WinGStretchBlt is optimized for copying WinGDCs to the screen.

Clipping can slow **WinGStretchBlt** down. In general, don’t select clipping regions into or blt outside the boundaries of the source or destination DCs and avoid blting to an overlapped window if possible.

See Also

WinGBitBlt WinGCreateDC WinGCreateBitmap WinGRecommendDIBFormat Maximizing Performance With WinG

WING_DITHER_TYPE

Dither types for halftone brushes.
WING_DITHER_TYPE

Values

DISPERSED_4x4
DISPERSED_8x8
CLUSTERED_4x4

See Also

WinGCreateHalftoneBrush WinGCreateHalftonePalette Halftoning With WinG CUBE

Debugging WinG Applications

WinG will report runtime errors and helpful debugging messages (for example, whether or not WinG has recognized an identity palette) through standard Windows methods (the serial port or applications such as DBWIN.EXE) if you so desire.

If you want WinG to send error messages to the debug output, make sure the following entry appears in your WIN.INI file. If there is already a [WinG] section, just add the Debug=1 line under that heading.

```
[WinG]
Debug=1
```

If you specifically do not want debug messages to appear, set this to:

```
[WinG]
Debug=0
```

If neither debug level is specified in the [WinG] section of your WIN.INI file, debugging will be turned ON if you're using the Windows debug kernel and OFF if you're using the Windows retail kernel. Setting the Debug level explicitly in your WIN.INI will always override this default behavior.

Shipping a Product With WinG

If your application uses WinG, you will have to copy the WinG runtime files into the \SYSTEM subdirectory of the Windows directory if WinG has not been previously installed on the target system. The following files should be installed on the user's system:

WING.DLL
WING32.DLL
DIBENG.DLL
WINGDIB.DRV
WINGPAL.WND

Microsoft will make the WinG libraries generally available to Windows developers for free distribution with Windows applications.

The Windows Software Development Kit includes the Setup Toolkit for Windows, which allows you to create and run setup scripts for installing Windows applications.

Documentation for the toolkit comes with the Windows SDK and is also available on the Microsoft Developer Network CD.

The WinG Development Kit setup program installs the WinG runtime files using Microsoft setup exactly as they should be installed on a target user's system. Look at the SETUP.MST script on the WinG installation diskette to see how this is done.

Code Samples

The WinG development kit contains a variety of code samples to help you develop fast applications quickly using WinG.

Snippets

The following code samples appear in this help file:

- Setting up an off-screen buffer with WinG.
- Calculating the memory address of a scanline in a WinGBitmap.
- Creating an Identity Palette.
- Clearing the System Palette.
- Maximizing palette availability using the SYSPAL_NOSTATIC setting.
- Copying a logical palette to a WinGBitmap color table.
- Matching an RGB color to a halftone palette entry.

Sample Applications

The WinG Development Kit also contains source code for several sample applications, installed in the \SAMPLES subdirectory. The following applications are available:

DOGGIE allows the user to drag a sprite around the screen with the mouse, demonstrating off-screen composition, dirty rectangle animation, and custom blt routines. Includes source code for a sample 8-bit DIB to 8-bit DIB blt with one transparent color.

CUBE displays a halftoned rotating cube in a window that the user can manipulate with the mouse. It demonstrates off-screen composition, double-buffering, and using the halftone palette and halftone brushes with GDI to draw into a WinGDC.

TIMEWING tests and compares blt speeds of existing GDI functions with the WinG blt function. This sample will give you an idea of how WinG blts will compare to standard GDI functions.

HALFTONE converts 24-bit RGB DIBs to 8-bit DIBs by dithering them to the WinG Halftone Palette. The source code implements a standard 8x8 dither and color matching to the halftone palette.

PALANIM performs simple palette animation with an identity palette using WinG. This application uses all of the sample code appearing in this help file.

Balloon Doggie Sample

The Balloon Doggie sample application, found in the SAMPLES\DOGGIE subdirectory of the WinG development kit, demonstrates a simple dirty rectangle animation system. It creates a WinGDC and a WinGBitmap, which it uses as an off-screen buffer, and uses WinGBitBlt to update the screen.

Balloon Doggie includes source code for TransparentDIBits (in TBLT.C and FAST32.ASM), a fast DIB-to-DIB blt with transparency. TransparentDIBits demonstrates the use of custom drawing routines with WinG to provide functions not present or unacceptably slow in GDI.

Note that DOGGIE.EXE requires MASM 5.1 to compile.

Spinning Cube Sample

The CUBE.EXE sample application, found in the SAMPLES\CUBE subdirectory of the WinG development kit, demonstrates the use of Halftoning to create the appearance of more than 256 colors on an 8-bit palletized display device. Using WinGCreateHalftonePalette and WinGCreateHalftoneBrush, the spinning cube application halftones the faces of the cube to create lighting effects.

The Spinning Cube sample uses a standard double buffering architecture using a WinGDC and a WinGBitmap. It creates a WinGDC when the application starts, then creates and selects appropriate WinGBitmaps on WM_SIZE messages to keep the off-screen buffer the same size as the window's client region.

When appropriate, the application uses the GDI Polygon function to draw into the off-screen buffer then calls WinGBitBlt to copy the buffer to the screen.

The CUBE sample uses a simple floating-point vector and camera C++ class library (in DUMB3D.HPP and DUMB3D.CPP) that can be used as a starting point by those interested in generating 3D graphics.

WinG Timing Sample

The timing sample, TIMEWING.EXE, found in the SAMPLES\TIMEWING subdirectory of the WinG development kit, times and compares the blt speeds of BitBlt, StretchDIBits, and WinGBitBlt. The application provides a summary you can use to compare the speeds of these techniques on various video configurations and a framework you can use for your own timing tests.

On most platforms, WinGBitBlt will perform favorably in comparison to BitBlt and will blow StretchDIBits away. SetDIBitsToDevice and StretchDIBits are essentially the same API, so this function is not timed.

Note that StretchDIBits and WinGBitBlt operate on device-independent bitmaps whereas BltBlt operates on device-specific bitmaps, which require no translation and can sometimes be stored in the local memory of the graphics card itself. For this reason, BitBlt usually runs at speeds approaching video memory bandwidth, which is the target speed for WinGBitBlt.

Also note that some drivers, such as the No 9GXE, "cheat" on their BitBlts by keeping the last blted image in card memory. If the image is blted again, the card uses the cached image instead of the memory image. This can result in misleading performance benchmarks unless a different image is blted at each frame.

WinG Halftoning Sample

HALFTONE.EXE, found in the SAMPLES\HALFTONE subdirectory of the WinG development kit, dithers 24-bit DIBs to the WinG Halftone Palette using an 8x8 ordered dither.

The main function, DibHalftoneDIB in HALFTONE.C, does the real work in the dithering. The process of calculating an ordered dither is too complex to describe here, but a description of the techniques involved can be found in "Computer Graphics: Principles and Practice" by Foley, van Dam, Feiner, and Hughes. See the Further Reading article for more information on this book.

The aWinGHalftoneTranslation array found in HTTABLES.C converts a 2.6-bit-per-pixel computed halftone index into an entry in the halftone palette. To calculate the nearest match of an RGB color to the halftone palette, HALFTONE uses the following formula:

```
HalftoneIndex = (Red / 51) + (Green / 51) * 6 + (Blue / 51) * 36;
HalftoneColorIndex = aWinGHalftoneTranslation [HalftoneIndex];
```

See also the documentation for the WinGCreateHalftoneBrush function and the Halftoning With WinG article.

WinG Palette Animation Sample

The PALANIM.EXE application, found in the SAMPLES\PALANIM subdirectory of the WinG development kit, performs simple palette animation using AnimatePalette and WinGSetDIBColorTable as described in the Palette Animation With WinG article.

PALANIM gives the user the option of using the static colors in the palette to create a 254-color ramp or a 236-color ramp in an identity palette for fast blting.

The PALANIM sample uses the code samples found in this help file to perform all of its WinG functions.

WinG Glossary

Bottom-Up DIB: A DIB in which the first scan line in memory corresponds to the bottommost scanline when the DIB is displayed. This is the standard Windows DIB format.
Color Table: The table of RGB color values referenced by an color-indexed DIB.
Dirty Rectangle Animation: A double-buffering technique in which only the areas on the screen which have changed are updated from frame to frame.
Double Buffering: An animation technique in which images are composed entirely off-screen then copied in whole or in part to the display.
Halftone Palette: An identity palette carefully filled with an array of colors optimized for dithering images to 8 bits per pixel.
Halftoning: A technique for simulating unavailable colors using special patterns of available colors. Also called dithering.
Identity Palette: A logical palette that is a 1:1 match to the system palette.
Logical Palette: A palette object created by an application using the CreatePalette function.
Palette: A table of RGB colors associated with a GDI Device Context.
Palette Animation: An animation technique in which palette entries are shifted to create the appearance of movement.
Static Colors: Reserved colors in the system palette that can never be changed by an application. Under normal circumstances, twenty colors are so reserved.
System Colors: The colors used by Windows to draw captions, menu bars, text, and other Windows display elements.
System Palette: A copy of the hardware device palette maintained by the Palette Manager.
Top-Down DIB: A DIB in which the first scan line in memory corresponds to the topmost scanline when the DIB is displayed.
WinGBitmap: A special HBITMAP with a DIB as its drawing surface created for use in a WinGDC.
WinGDC: A device context with a DIB as its drawing surface.

Further Reading

The following collection of books, articles, and sample code may help clarify the use of DIBs, provide insight into custom drawing routines, or generally ease the transition from device-dependent bitmaps to WinGDCs. All of these are available on the Microsoft Developer Network CD. Some are included with the Windows SDK.

Foley, vanDam, Feiner, and Hughes, Computer Graphics: Principles and Practice, Second Edition, Addison-Wesley, 1991

Gery, Ron, “Using DIBs with Palettes,” Microsoft Technical Article, 3 March 1992

Gery, Ron, “DIBs and Their Use,” Microsoft Technical Article, 20 March 1992

Gery, Ron, “The Palette Manager: How and Why,” Microsoft Technical Article, 23 March 1992

Petzold, Charles, “The Device-Independent Bitmap (DIB),” Programming Windows 3.1, Microsoft Press, 1992, pp. 607-619

Rodent, Herman, “Animation In Windows,” Microsoft Technical Article, 28 April 1993

“How To Use a DIB Stored as a Windows Resource,” Microsoft PSS Article Q67883, 26 April 1993

“Multimedia Video Techniques,” Microsoft Technical Article, 20 March 1992

Windows 3.1 Software Development Kit samples: DIBIT, DIBVIEW, CROPDIB, WINCAP, SHOWDIB, FADE

Microsoft Technical Samples, TRIQ, draws triangles or boxes directly into device-independent bitmap memory.