

DirectDraw

This section provides information about the DirectDraw® component of the DirectX® application programming interface (API). Information is divided into the following groups:

- About DirectDraw
- Why Use DirectDraw?
- Getting Started: Basic Graphics Concepts
- DirectDraw Architecture
- DirectDraw Essentials
- DirectDraw Tutorials
- DirectDraw Reference
- DirectDraw Samples

About DirectDraw

DirectDraw® is the component of the DirectX® application programming interface (API) that allows you to directly manipulate display memory, the hardware blitter, hardware overlay support, and flipping surface support. DirectDraw provides this functionality while maintaining compatibility with existing Microsoft® Windows®-based applications and device drivers.

DirectDraw is a software interface that provides direct access to display devices while maintaining compatibility with the Windows graphics device interface (GDI). It is not a high-level application programming interface (API) for graphics. DirectDraw provides a device-independent way for games and Windows subsystem software, such as three-dimensional (3-D) graphics packages and digital video codecs, to gain access to the features of specific display devices.

DirectDraw works with a wide variety of display hardware, ranging from simple SVGA monitors to advanced hardware implementations that provide clipping, stretching, and non-RGB color format support. The interface is designed so that your applications can enumerate the capabilities of the underlying hardware and then use any supported hardware-accelerated features. Features that are not implemented in hardware are emulated by DirectX.

DirectDraw provides device-dependent access to display memory in a device-independent way. Essentially, DirectDraw manages display memory. Your application need only recognize some basic device dependencies that are standard across hardware implementations, such as RGB and YUV color formats and the pitch between raster lines. You need not call specific procedures to use the blitter or manipulate palette registers. Using DirectDraw, you can manipulate display memory with ease, taking full advantage of the blitting and color decompression capabilities of different types of display hardware without becoming dependent on a particular piece of hardware.

DirectDraw provides world-class game graphics on computers running Windows 95 and later and Windows NT® version 4.0 or Windows 2000.

Why Use DirectDraw?

The DirectDraw component brings many powerful features to you, the Windows graphics programmer:

- The hardware abstraction layer (HAL) of DirectDraw provides a consistent interface through which to work directly with the display hardware, getting maximum performance.
- DirectDraw assesses the video hardware's capabilities, making use of special hardware features whenever possible. For example, if your video card supports hardware blitting, DirectDraw delegates blits to the video card, greatly increasing performance. Additionally, DirectDraw provides a hardware emulation layer (HEL) to support features when the hardware does not.
- DirectDraw exists under Windows, gaining the advantage of 32-bit memory addressing and a flat memory model that the operating system provides. DirectDraw presents video and system memory as large blocks of storage, not as small segments. If you've ever used segment:offset addressing, you will quickly begin to appreciate this "flat" memory model.
- DirectDraw makes it easy for you to implement page flipping with multiple back buffers in full-screen applications. For more information, see Page Flipping and Back Buffering.

- Support for clipping in windowed or full-screen applications.
- Support for 3-D z-buffers.
- Support for hardware-assisted overlays with z-ordering.
- Access to image-stretching hardware.
- Simultaneous access to standard and enhanced display-device memory areas.
- Other features include custom and dynamic palettes, exclusive hardware access, and resolution switching.

These features combine to make it possible for you to write applications that easily outperform standard Windows GDI-based applications and even MS-DOS applications.

Getting Started: Basic Graphics Concepts

This section provides an overview of graphics programming with DirectDraw. Each concept discussed here begins with a non-technical overview, followed by some specific information about how DirectDraw supports it.

You don't need to be a graphics guru to benefit from this overview—in fact, if you are one you might want to skip this section entirely and move on to the more detailed information in the DirectDraw Essentials section. If you're familiar with Windows programming in C and C++, you won't have difficulty digesting this information. When you finish reading these topics, you will have a solid understanding of basic DirectDraw graphics programming concepts.

The following topics are discussed:

- Device-Independent Bitmaps
- Drawing Surfaces
- Blitting
- Page Flipping and Back Buffering
- Introduction to Rectangles

Device-Independent Bitmaps

Windows, and therefore DirectX, uses the device-independent bitmap (DIB) as its native graphics file format. Essentially, a DIB is a file that contains information describing an image's dimensions, the number of colors it uses, values describing those colors, and data that describes each pixel. Additionally, a DIB contains some lesser-used parameters, like information about file compression, significant colors (if all are not used), and physical dimensions of the image (in case it will end up in print). DIB files usually have the .bmp file extension, although they might occasionally have a .dib extension.

Because the DIB is so pervasive in Windows programming, the Platform SDK already contains many functions that you can use with DirectX. For example, the following application-defined function, taken from the Ddutil.cpp file that comes with the DirectX APIs in the Platform SDK, combines Win32® and DirectX functions to load a DIB onto a DirectX surface.

```
extern "C" IDirectDrawSurface * DDLoadBitmap(IDirectDraw *pdd,
    LPCSTR szBitmap, int dx, int dy)
{
    HBITMAP          hbm;
    BITMAP           bm;
    DDSURFACEDESC    ddsd;
    IDirectDrawSurface *pdds;

    //
    // This is the Win32 part.
    // Try to load the bitmap as a resource.
    // If that fails, try it as a file.
    //
    hbm = (HBITMAP) LoadImage(
        GetModuleHandle(NULL), szBitmap,
        IMAGE_BITMAP, dx, dy, LR_CREATEDIBSECTION);

    if (hbm == NULL)
```

```

    hbm = (HBITMAP) LoadImage(
        NULL, szBitmap, IMAGE_BITMAP, dx, dy,
        LR_LOADFROMFILE|LR_CREATEDIBSECTION);

    if (hbm == NULL)
        return NULL;

    //
    // Get the size of the bitmap.
    //
    GetObject(hbm, sizeof(bm), &bm);

    //
    // Now, return to DirectX function calls.
    // Create a DirectDrawSurface for this bitmap.
    //
    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    ddsd.dwWidth = bm.bmWidth;
    ddsd.dwHeight = bm.bmHeight;

    if (pdd->CreateSurface(&ddsd, &pdds, NULL) != DD_OK)
        return NULL;

    DDCopyBitmap(pdds, hbm, 0, 0, 0, 0);

    DeleteObject(hbm);

    return pdds;
}

```

For more detailed information about DIB files, see the Platform SDK.

Drawing Surfaces

Drawing surfaces receive video data to eventually be displayed on the screen as images (bitmaps, to be exact). In most Windows programs, you get access to the drawing surface using a Win32 function such as **GetDC**, which stands for get the device context (DC). After you have the device context, you can start painting the screen. However, Win32 graphics functions are provided by an entirely different part of the system, the graphics device interface (GDI). The GDI is a system component that provides an abstraction layer that enables standard Windows applications to draw to the screen.

The drawback of GDI is that it wasn't designed for high-performance multimedia software, it was made to be used by business applications like word processors and spreadsheet applications. GDI provides access to a video buffer in system memory, not video memory, and doesn't take advantage of special features that some video cards provide. In short, GDI is great for most business applications, but its performance is too slow for multimedia or game software.

On the other hand, DirectDraw can give you drawing surfaces that represent actual video memory. This means that when you use DirectDraw, you can write directly to the memory on the video card, making your graphics routines extremely fast. These surfaces are represented as contiguous blocks of memory, making it easy to perform addressing within them.

For more detailed information, see Surfaces.

Blitting

The term *blit* is shorthand for "bit block transfer," which is the process of transferring blocks of data from one place in memory to another. Graphics programmers use blitting to transfer graphics from one place in memory to another. Blits are often used to perform sprite animation, which is discussed later.

For more information on blitting in DirectDraw, see [Blitting to Surfaces](#).

Page Flipping and Back Buffering

Page flipping is key in multimedia, animation, and game software. Software page flipping is analogous to the way animation can be done with a pad of paper. On each page the artist changes the figure slightly, so that when you flip between sheets rapidly the drawing appears animated.

Page flipping in software is very similar to this process. Initially, you set up a series of DirectDraw surfaces that are designed to "flip" to the screen the way artist's paper flips to the next page. The first surface is referred to as the primary surface, and the surfaces behind it are called back buffers. Your application writes to a back buffer, then flips the primary surface so that the back buffer appears on screen. While the system is displaying the image, your software is again writing to a back buffer. The process continues as long as you're animating, allowing you to animate images quickly and efficiently.

DirectDraw makes it easy for you to set up page flipping schemes, from a relatively simple double-buffered scheme (a primary surface with one back buffer) to more sophisticated schemes that add additional back buffers. For more information see [DirectDraw Tutorials](#) and [Flipping Surfaces](#).

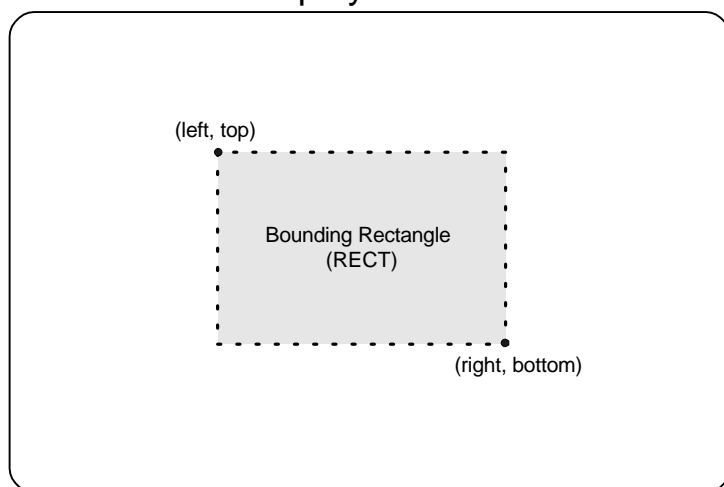
Introduction to Rectangles

Throughout DirectDraw and Windows programming, objects on the screen are referred to in terms of bounding rectangles. The sides of a bounding rectangle are always parallel to the sides of the screen, so the rectangle can be described by two points, the top-left corner and bottom-right corner. Most applications use the **RECT** structure to carry information about a bounding rectangle to use when blitting to the screen or performing hit detection. The **RECT** structure has the following definition:

```
typedef struct tagRECT {
    LONG   left;    // This is the top-left corner's x-coordinate.
    LONG   top;     // The top-left corner's y-coordinate.
    LONG   right;   // The bottom-right corner's x-coordinate.
    LONG   bottom;  // The bottom-right corner's y-coordinate.
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

In the preceding example, the **left** and **top** members are the x- and y-coordinates of a bounding rectangle's top-left corner. Similarly, the **right** and **bottom** members make up the coordinates of the bottom-right corner. The following diagram illustrates how you can visualize these values.

Display Screen



In the interest of efficiency, consistency, and ease of use, all DirectDraw blitting functions work with rectangles. However, you can create the illusion of nonrectangular blit operations by using transparent blitting. For more information, see [Transparent Blitting](#).

DirectDraw Architecture

This section contains general information about the relationship between the DirectDraw component and the rest of DirectX, the operating system, and the system hardware. The following topics are discussed:

- Architectural Overview for DirectDraw
- DirectDraw Object Types
- Hardware Abstraction Layer (HAL)
- Software Emulation
- System Integration

Architectural Overview for DirectDraw

Multimedia software requires high-performance graphics. Through DirectDraw, Microsoft enables a much higher level of efficiency and speed in graphics-intensive applications for Windows than is possible with GDI, while maintaining device independence. DirectDraw provides tools to perform such key tasks as:

- Manipulating multiple display surfaces
- Accessing the video memory directly
- Page flipping
- Back buffering
- Managing the palette
- Clipping

Additionally, DirectDraw enables you to query the display hardware's capabilities at run time, then provide the best performance possible given the host computer's hardware capabilities.

As with other DirectX components, DirectDraw uses the hardware to its greatest possible advantage, and provides software emulation for most features when hardware support is unavailable. Device independence is possible through use of the hardware abstraction layer, or HAL. For more information about the HAL, see the hardware abstraction layer .

The DirectDraw component provides services through COM-based interfaces. In the most recent iteration, these interfaces are **IDirectDraw4**, **IDirectDrawSurface4**, **IDirectDrawPalette**, **IDirectDrawClipper**, and **IDirectDrawVideoPort**. Note that, in addition to these interfaces, DirectDraw continues to support all previous versions. The DirectDraw component doesn't expose an **IDirectDraw3** interface, the interface versions skipped from **IDirectDraw2** to **IDirectDraw4**.

For more information about COM concepts that you should understand to create applications with the DirectX APIs in the Platform SDK, see DirectX and the Component Object Model.

The DirectDraw object represents the display adapter and exposes its methods through the **IDirectDraw**, **IDirectDraw2**, and **IDirectDraw4** interfaces. In most cases you will use the **DirectDrawCreate** function to create a DirectDraw object, but you can also create one with the **CoCreateInstance** COM function. For more information, see Creating DirectDraw Objects by Using CoCreateInstance.

After creating a DirectDraw object, you can create surfaces for it by calling the **IDirectDraw4::CreateSurface** method. Surfaces represent the memory on the display hardware, but can exist on either video memory or system memory. DirectDraw extends support for palettes, clipping (useful for windowed applications), and video ports through its other interfaces.

DirectDraw Object Types

You can think of DirectDraw as being composed of several objects that work together. This section briefly describes the objects you use when working with the DirectDraw component, organized by object type. For detailed information, see DirectDraw Essentials.

The DirectDraw component uses the following objects:

DirectDraw object

The DirectDraw object is the heart of all DirectDraw applications. It's the first object you create, and you use it to make all other related objects. You create a DirectDraw object by calling the **DirectDrawCreate** function. DirectDraw objects expose

their functionality through the **IDirectDraw**, **IDirectDraw2**, and **IDirectDraw4** interfaces. For more information, see The DirectDraw Object.

DirectDrawSurface object

The DirectDrawSurface object (casually referred to as a "surface") represents an area in memory that holds data to be displayed on the monitor as images or moved to other surfaces. You usually create a surface by calling the **IDirectDraw4::CreateSurface** method of the DirectDraw object with which it will be associated. DirectDrawSurface objects expose their functionality through the **IDirectDrawSurface**, **IDirectDrawSurface2**, **IDirectDrawSurface3**, and **IDirectDrawSurface4** interfaces. For more information, see Surfaces.

DirectDrawPalette object

The DirectDrawPalette object (casually referred to as a "palette") represents a 16- or 256-color indexed palette to be used with a surface. It contains a series of indexed RGB triplets that describe colors associated with values within a surface. You do not use palettes with surfaces that use a pixel format depth greater than 8 bits. You can create a DirectDrawPalette object by calling the **IDirectDraw4::CreatePalette** method. DirectDrawPalette objects expose their functionality through the **IDirectDrawPalette** interface. For more information, see Palettes.

DirectDrawClipper object

The DirectDrawClipper object (casually referred to as a "clipper") helps you prevent blitting to certain portions of a surface or beyond the bounds of a surface. You can create a clipper by calling the **IDirectDraw4::CreateClipper** method. DirectDrawClipper objects expose their functionality through the **IDirectDrawClipper** interface. For more information, see Clippers.

DirectDrawVideoPort object

The DirectDrawVideoPort object represents video-port hardware present in some systems. This hardware allows direct access to the frame buffer without accessing the CPU or using the PCI bus. You can create a DirectDrawVideoPort object by calling a **QueryInterface** method for the DirectDraw object, specifying the IID_IDDVideoPortContainer reference identifier. DirectDrawVideoPort objects expose their functionality through the **IDDVideoPortContainer** and **IDirectDrawVideoPort** interfaces. For more information, see Video Ports.

Hardware Abstraction Layer (HAL)

DirectDraw provides device independence through the hardware abstraction layer (HAL). The HAL is a device-specific interface, provided by the device manufacturer, that DirectDraw uses to work directly with the display hardware. Applications never interact with the HAL. Rather, with the infrastructure that the HAL provides, DirectDraw exposes a consistent set of interfaces and methods that an application uses to display graphics. The device manufacturer implements the HAL in a combination of 16-bit and 32-bit code under Windows. Under Windows NT/Windows 2000, the HAL is always implemented in 32-bit code. The HAL can be part of the display driver or a separate DLL that communicates with the display driver through a private interface that driver's creator defines.

The DirectDraw HAL is implemented by the chip manufacturer, board producer, or original equipment manufacturer (OEM). The HAL implements only device-dependent code and performs no emulation. If a function is not performed by the hardware, the HAL does not report it as a hardware capability. Additionally, the HAL does not validate parameters; DirectDraw does this before the HAL is invoked.

Software Emulation

When the hardware does not support a feature through the hardware abstraction layer (HAL), DirectDraw attempts to emulate it. This emulated functionality is provided through the hardware emulation layer (HEL). The HEL presents its capabilities to DirectDraw just as the HAL would. And, as with the HAL, applications never work directly with the HEL. The result is transparent support for almost all major features, regardless of whether a given feature is supported by hardware or through the HEL.

Obviously, software emulation cannot equal the performance that hardware features provide. You can query for the features the hardware supports by using the **IDirectDraw4::GetCaps** method. By examining these capabilities during application initialization, you can adjust application parameters to provide optimum performance over varying levels of hardware performance.

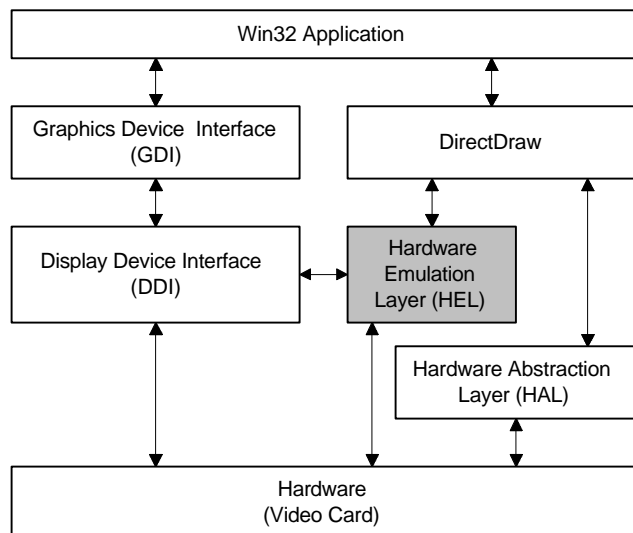
In some cases, certain combinations of hardware supported features and emulation can result in slower performance than emulation alone. For example, if the display device driver supports DirectDraw but not stretch blitting, noticeable performance losses will occur when stretch blitting from video memory surfaces. This happens because video memory is often slower than system memory, forcing the CPU to wait when accessing video memory surfaces. If your application uses a feature that isn't

supported by the hardware, it is sometimes best to create surfaces in system memory, thereby avoiding performance losses created when the CPU accesses video memory.

For more information, see Hardware Abstraction Layer (HAL).

System Integration

The following diagram shows the relationships between DirectDraw, the graphics device interface (GDI), the hardware abstraction layer (HAL), the hardware emulation layer (HEL) and the hardware.



As the preceding diagram shows, a DirectDraw object exists alongside GDI, and both have direct access to the hardware through a device-dependent abstraction layer. Unlike GDI, DirectDraw makes use of special hardware features whenever possible. If the hardware does not support a feature, DirectDraw attempts to emulate it by using the HEL. DirectDraw can provide surface memory in the form of a device context, making it possible for you to use GDI functions to work with surface objects.

DirectDraw Essentials

This section contains general information about the DirectDraw® component of DirectX®. Information is organized into the following groups:

- Cooperative Levels
- Display Modes
- The DirectDraw Object
- Surfaces
- Palettes
- Clippers
- Multiple Monitor Systems
- Advanced DirectDraw Topics

Cooperative Levels

In the following topics, this section introduces the concept of cooperative levels and describes some common usage situations:

- About Cooperative Levels
- Testing Cooperative Levels

About Cooperative Levels

Cooperative levels describe how DirectDraw interacts with the display and how it reacts to events that might affect the display. Use the **IDirectDraw4::SetCooperativeLevel** method to set cooperative level of DirectDraw. For the most part, you use DirectDraw cooperative levels to determine whether your application runs as a full-screen program with exclusive access to the display or as a windowed application. However, DirectDraw cooperative levels can also have the following effects:

- Enable DirectDraw to use Mode X resolutions. For more information, see Mode X and Mode 13 Display Modes.
- Prevent DirectDraw from releasing exclusive control of the display or rebooting if the user presses CTRL + ALT + DEL (exclusive mode only).
- Enable DirectDraw to minimize or maximize the application in response to activation events.

The normal cooperative level indicates that your DirectDraw application will operate as a windowed application. At this cooperative level you won't be able to change the primary surface's palette or perform page flipping.

Because applications can use DirectDraw with multiple windows, **IDirectDraw4::SetCooperativeLevel** does not require a window handle to be specified if the application is requesting the DDSCL_NORMAL mode. By passing a NULL to the window handle, all of the windows can be used simultaneously in normal Windows mode.

At the full-screen and exclusive cooperative level, you can use the hardware to its fullest. In this mode, you can set custom and dynamic palettes, change display resolutions, and implement page flipping. The exclusive (full-screen) mode does not prevent other applications from allocating surfaces, nor does it exclude them from using DirectDraw or GDI. However, it does prevent applications other than the one currently with exclusive access from changing the display mode or palette.

DirectDraw takes control of window activation events for full-screen, exclusive mode applications, sending WM_ACTIVATEAPP messages to the window handle registered through the **SetCooperativeLevel** method as needed. DirectDraw only sends activation events to the top-level window. If your application creates child windows that require activation event messages, it is your responsibility to subclass the child windows.

SetCooperativeLevel maintains a binding between a process and a window handle. If **SetCooperativeLevel** is called once in a process, a binding is established between the process and the window. If it is called again in the same process with a different non-null window handle, it returns the DDERR_HWNDALREADYSET error value. Some applications may receive this error value when DirectSound® specifies a different window handle than DirectDraw—they should specify the same, top-level application window handle.

Note

Developers using Microsoft Foundation Classes (MFC) should keep in mind that the window handle given to the **SetCooperativeLevel** method should identify the application's top-level window, not a derived child window. To retrieve your MFC application's top level window handle, you could use the following code:

```
HWND hwndTop = AfxGetMainWnd()->GetSafeHwnd();
```

See also, Multiple Monitor Systems.

Testing Cooperative Levels

Developers often use messages such as WM_ACTIVATEAPP and WM_DISPLAYCHANGE as notifications that their applications should restore or re-create the surfaces being used. In some cases, applications take action when they don't need to, or don't take action when they should. The **IDirectDraw4::TestCooperativeLevel** method makes it possible for your application to retrieve more information about the DirectDraw object's cooperative level and take appropriate steps to continue execution without mishap.

The **TestCooperativeLevel** method succeeds, returning DD_OK, if your application can restore its surfaces (if it has not already done so) and continue to execute. Failure codes, on the other hand, are interpreted differently depending on the cooperative-level your application uses:

Full-screen applications

Full-screen applications receive the DDERR_NOEXCLUSIVEMODE return value if they lose exclusive device access—for example, if the user pressed ALT+TAB to switch away from the current application. In this case, applications might call **TestCooperativeLevel** in a loop, exiting only when the method returns DD_OK (meaning that exclusive mode was returned). In the body of the loop, the application should relinquish control of the CPU to prevent using cycles unnecessarily. Windows supports functions such as the **WaitMessage** or **Sleep** Win32 functions for this purpose.

Any existing surfaces should be restored by calling the **IDirectDrawSurface4::Restore** or **IDirectDraw4::RestoreAllSurfaces** methods, and their contents reloaded before displaying them.

Windowed applications

Windowed applications (those that use the normal cooperative level) receive **DDERR_EXCLUSIVEMODEALREADYSET** if another application has taken exclusive device access. In this case, no action should be taken until the application with exclusive access loses it. This situation is similar to the case for a full-screen application; a windowed application might loop until **TestCooperativeLevel** returns **DD_OK** before restoring and reloading its surfaces. As mentioned previously, in a loop like this applications should avoid unnecessarily using CPU cycles by relinquishing CPU control periodically during the loop.

The **TestCooperativeLevel** method returns **DDERR_WRONGMODE** to windowed applications when the display mode has changed. In this case, the application should destroy and re-create any surfaces before continuing execution.

Display Modes

This section contains general information about DirectDraw display modes. The following topics are discussed:

- About Display Modes
- Determining Supported Display Modes
- Setting Display Modes
- Restoring Display Modes
- Mode X and Mode 13 Display Modes
- Support for High Resolutions and True-Color Bit Depths

About Display Modes

A display mode is a hardware setting that describes the dimensions and bit-depth of graphics that the display hardware sends to the monitor from the primary surface. Display modes are described by their defining characteristics: width, height, and bit-depth. For instance, most display adapters can display graphics 640 pixels wide and 480 pixels tall, where each pixel is 8 bits of data. In shorthand, this display mode is called 640×480×8. As the dimensions of a display mode get larger or as the bit-depth increases, more display memory is required.

There are two types of display modes: palettized and non-palettized. For palettized display modes, each pixel is a value representing an index into an associated palette. The bit depth of the display mode determines the number of colors that can be in the palette. For instance, in an 8-bit palettized display mode, each pixel is a value from 0 to 255. In such a display mode, the palette can contain 256 entries.

Non-palettized display modes, as their name states, do not use palettes. The bit depth of a non-palettized display mode indicates the total number of bits that are used to describe a pixel.

The primary surface and any surfaces in the primary flipping chain match the display mode's dimensions, bit depth and pixel format. For more information, see Pixel Formats.

Determining Supported Display Modes

Because display hardware varies, not all devices will support all display modes. To determine the display modes supported on a given system, call the **IDirectDraw4::EnumDisplayModes** method. By setting the appropriate values and flags, the **EnumDisplayModes** method can list all supported display modes or confirm that a single display mode that you specify is supported. The method's first parameter, *dwFlags*, controls extra options for the method; in most cases, you will set *dwFlags* to 0 to ignore extra options. The second parameter, *lpDDSurfaceDesc*, is the address of a **DDSURFACEDESC2** structure that describes a given display mode to be confirmed; you'll usually set this parameter to NULL to request that all modes be listed. The third parameter, *lpContext*, is a pointer that you want DirectDraw to pass to your callback function; if you don't need any extra data in the callback function, use NULL here. Last, you set the *lpEnumModesCallback* parameter to the address of the callback function that DirectDraw will call for each supported mode.

The callback function you supply when calling **EnumDisplayModes** must match the prototype for the **EnumModesCallback** function. For each display mode that the hardware supports, DirectDraw calls your callback function passing two parameters. The first parameter is the address of a **DDSURFACEDESC2** structure that describes one supported display mode, and the second parameter is the address of the application-defined data you specified when calling **EnumDisplayModes**, if any.

Examine the values in the **DDSURFACEDESC2** structure to determine the display mode it describes. The key structure members are the **dwWidth**, **dwHeight**, and **ddpfPixelFormat** members. The **dwWidth** and **dwHeight** members describe the display mode's dimensions, and the **ddpfPixelFormat** member is a **DDPIXELFORMAT** structure that contains information about the mode's bit depth.

The **DDPIXELFORMAT** structure carries information describing the mode's bit depth and tells you whether or not the display mode uses a palette. If the **dwFlags** member contains the **DDPF_PALETTEINDEXED1**, **DDPF_PALETTEINDEXED2**, **DDPF_PALETTEINDEXED4**, or **DDPF_PALETTEINDEXED8** flag, the display mode's bit depth is 1, 2, 4 or 8 bits, and each pixel is an index into an associated palette. If **dwFlags** contains **DDPF_RGB**, then the display mode is non-palettized and its bit depth is provided in the **dwRGBBitCount** member of the **DDPIXELFORMAT** structure.

Setting Display Modes

You can set the display mode by using the **IDirectDraw4::SetDisplayMode** method. The **SetDisplayMode** method accepts four parameters that describe the dimensions, bit depth, and refresh rate of the mode to be set. The method uses a fifth parameter to indicate special options for the given mode; this is currently only used to differentiate between Mode 13 and the Mode X 320×200×8 display mode.

Although you can specify the desired display mode's bit depth, you cannot specify the pixel format that the display hardware will use for that bit depth. To determine the RGB bit masks that the display hardware uses for the current bit depth, call **IDirectDraw4::GetDisplayMode** after setting the display mode. If the current display mode is not palettized, you can examine the mask values in the **dwRBitMask**, **dwGBitMask**, and **dwBBitMask** members to determine the correct red, green, and blue bits. For more information, see Pixel Format Masks.

Modes can be changed by more than one application as long as they are all sharing a display card. You can change the bit depth of the display mode only if your application has exclusive access to the DirectDraw object. All DirectDrawSurface objects lose surface memory and become inoperative when the mode is changed. A surface's memory must be reallocated by using the **IDirectDrawSurface4::Restore** method.

The DirectDraw exclusive (full-screen) mode does not bar other applications from allocating DirectDrawSurface objects, nor does it exclude them from using DirectDraw or GDI functionality. However, it does prevent applications other than the one that obtained exclusive access from changing the display mode or palette.

Note

You can only call the **IDirectDraw4::SetDisplayMode** method from the thread that created the application window. For single threaded applications (the vast majority), this restriction isn't an issue.

Restoring Display Modes

You can explicitly restore the display hardware to its original mode by calling the **IDirectDraw4::RestoreDisplayMode** method. If the display mode was set by calling **IDirectDraw4::SetDisplayMode** and your application takes the exclusive cooperative level, the original display mode is reset automatically when you set the application's cooperative level back to normal. (This behavior was first offered in the **IDirectDraw2** interface, and is offered by all newer versions of the interface.)

If you're using the **IDirectDraw** interface, you must always explicitly restore the display mode by using the **RestoreDisplayMode** method.

Mode X and Mode 13 Display Modes

DirectDraw supports both Mode 13 and Mode X display modes. Mode 13 is the linear unflippable 320×200×8 bits per pixel palettized mode known widely by its hexadecimal BIOS mode number: 13. For more information, see Mode 13 Support. Mode X is a hybrid display mode derived from the standard VGA Mode 13. This mode allows the use of up to 256 kilobytes (KB) of display memory (rather than the 64 KB allowed by Mode 13) by using the VGA display adapter's EGA multiple video plane system.

DirectDraw provides two Mode X modes (320×200×8 and 320×240×8) for all display cards. Some cards also support linear low-resolution modes. In linear low-resolution modes, the primary surface can be locked and directly accessed. This is not possible in Mode X modes.

Mode X modes are available only if an application uses the **DDSCL_ALLOWMODEX**, **DDSCL_FULLSCREEN**, and **DDSCL_EXCLUSIVE** flags when calling the **IDirectDraw4::SetCooperativeLevel** method. If **DDSCL_ALLOWMODEX** is not

specified, the **IDirectDraw4::EnumDisplayModes** method will not enumerate Mode X modes, and the **IDirectDraw4::SetDisplayMode** method will fail if a Mode X mode is requested.

Windows 95 and Windows NT/Windows 2000 do not natively support Mode X modes; therefore, when your application is in a Mode X mode, you cannot use the **IDirectDrawSurface4::Lock** or **IDirectDrawSurface4::Blt** methods to lock or blit to the primary surface. You also cannot use either the **IDirectDrawSurface4::GetDC** method on the primary surface, or GDI with a screen DC. Mode X modes are indicated by the DDSCAPS_MODEX flag in the **DDSCAPS2** structure, which is part of the **DDSURFACEDESC2** structure returned by the **IDirectDrawSurface4::GetCaps** and **IDirectDraw4::EnumDisplayModes** methods.

Support for High Resolutions and True-Color Bit Depths

DirectDraw supports all of the screen resolutions and depths supported by the display device driver. DirectDraw allows an application to change the mode to any one supported by the computer's display driver, including all supported 24- and 32-bpp (true-color) modes.

DirectDraw also supports HEL blitting in true-color surfaces. If the display device driver supports blitting at these resolutions, the hardware blitter will be used for display-memory-to-display-memory blits. Otherwise, the HEL will be used to perform the blits.

DirectDraw checks a list of known display modes against the display restrictions of the installed monitor. If DirectDraw determines that the requested mode is not compatible with the monitor, the call to the **IDirectDraw4::SetDisplayMode** method fails. Only modes that are supported on the installed monitor will be enumerated when you call the **IDirectDraw4::EnumDisplayModes** method.

The DirectDraw Object

This section contains information about DirectDraw objects and how you can manipulate them through their **IDirectDraw**, **IDirectDraw2**, or **IDirectDraw4** interfaces. The following topics are discussed:

- What Are DirectDraw Objects?
- What's New in IDirectDraw4?
- Parent and Child Object Lifetimes
- Multiple DirectDraw Objects per Process
- Creating DirectDraw Objects by Using CoCreateInstance

What Are DirectDraw Objects?

The DirectDraw object is the heart of all DirectDraw applications and is an integral part of Direct3D® applications as well. It is the first object you create and, through it, you create all other related objects. Typically, you create a DirectDraw object by calling the **DirectDrawCreate** function, which returns an **IDirectDraw** interface. If you want to work with a different iteration of the interface (such as **IDirectDraw4**) to take advantage of new features it provides, you can query for it. (See Getting an IDirectDraw4 Interface.) Note that you can create multiple DirectDraw objects, one for each display device installed in a system.

The DirectDraw object represents the display device and makes use of hardware acceleration if the display device for which it was created supports hardware acceleration. Each unique DirectDraw object can manipulate the display device and create surfaces, palettes, and clipper objects that are dependent on (or are, "connected to") the object that created them. For example, to create surfaces, you call the **IDirectDraw4::CreateSurface** method. Or, if you need a palette object to apply to a surface, call the **IDirectDraw4::CreatePalette** method. Additionally, the **IDirectDraw4** interface exposes similar methods to create clipper objects.

You can create more than one instance of a DirectDraw object at a time. The simplest example of this is using two monitors on a Windows 95 or Windows NT 4.0 and earlier system. Although these operating systems don't support dual monitors on their own, it is possible to write a DirectDraw HAL for each display device. The display device Windows and GDI recognizes is the one that will be used when you create the instance of the default DirectDraw object. The display device that Windows and GDI do not recognize can be addressed by another, independent DirectDraw object that must be created by using the second display device's globally unique identifier (GUID). This GUID can be obtained by using the **DirectDrawEnumerate** function.

The DirectDraw object manages all of the objects it creates. It controls the default palette (if the primary surface is in 8-bits-per-pixel mode), the default color key, and the hardware display mode. It tracks what resources have been allocated and what resources remain to be allocated.

What's New in IDirectDraw4?

This section details new features provided by the **IDirectDraw4** interface and describes its new features or how it behaves differently than its predecessor, **IDirectDraw2** (there is no **IDirectDraw3** interface). The following topics are discussed:

- New Features in IDirectDraw4
- Getting an IDirectDraw4 Interface

The most obvious difference between the **IDirectDraw4** interface and its predecessors is how it works with surfaces—how surfaces are described and which interfaces it automatically provides to access them. All of the surface-related methods in the new interface accept slightly different parameters than their counterparts in former interface versions. Wherever an **IDirectDraw2** interface method might accept a **DDSURFACEDESC** structure or retrieve an **IDirectDrawSurface3** interface, the methods of **IDirectDraw4** accept a **DDSURFACEDESC2** structure and retrieve an **IDirectDrawSurface4** interface instead.

Another behavioral change that **IDirectDraw4** introduces affects the lifetimes of child objects with respect to their parent DirectDraw object. For more information, see Parent and Child Object Lifetimes.

New Features in IDirectDraw4

The **IDirectDraw4** interface extends previous iterations by adding several methods that provide improved surface management and ease of use.

The **IDirectDraw4** interface exposes the new **IDirectDraw4::RestoreAllSurfaces** method, which restores all of the surfaces created by a DirectDraw with a single call.

Additionally, you can now retrieve a surface's **IDirectDrawSurface4** interface from a Windows device context by using the **IDirectDraw4::GetSurfaceFromDC** method.

Getting an IDirectDraw4 Interface

The Component Object Model on which DirectX is built specifies that an object can provide new functionality through new interfaces, without affecting backward compatibility. To this end, the **IDirectDraw4** interface supersedes the **IDirectDraw2** interface. This new interface can be obtained by using the **IUnknown::QueryInterface** method, as the following C++ example shows:

```
// Create an IDirectDraw4 interface.
LPDIRECTDRAW 1pDD;
LPDIRECTDRAW4 1pDD4;

ddrval = DirectDrawCreate(NULL, &1pDD, NULL);
if(ddrval != DD_OK)
    return;

ddrval = 1pDD->SetCooperativeLevel(hwnd,
    DDSCL_NORMAL);
if(ddrval != DD_OK)
    return;

ddrval = 1pDD->QueryInterface(IID_IDirectDraw4,
    (LPVOID *)&1pDD4);
if(ddrval != DD_OK)
    return;
```

The preceding example creates a DirectDraw object, then calls the **IUnknown::QueryInterface** method of the **IDirectDraw** interface it received to create an **IDirectDraw4** interface.

After getting an **IDirectDraw4** interface, you can begin calling its methods to take advantage of new features, performance improvements, and behavioral differences. Because some methods might change with the release of a new interface, mixing methods from an interface and its replacement (between **IDirectDraw2** and **IDirectDraw4**, for example) can cause unpredictable results.

Parent and Child Object Lifetimes

All objects you'll use in DirectDraw programming—the DirectDraw object, surfaces, palettes, clippers, and such—only exist in memory for as long as another object, such as an application, needs them. The time that passes from the moment when an object is created and placed in memory to when it is released and subsequently removed from memory is known as the object's lifetime. The Component Object Model (COM) followed by all DirectX components dictates that an object must keep track of how many other objects require its services. This number, known as a reference count, determines the object's lifetime. COM also dictates that an object expose the **IUnknown::AddRef** and **IUnknown::Release** methods to enable applications to explicitly manage its reference count; make sure you use these methods in accordance to COM rules.

You aren't the only one who is using the **IUnknown** methods to manage reference counts for objects—DirectDraw objects use them internally, too. When you use the **IDirectDraw4** interface (in contrast to **IDirectDraw2** or **IDirectDraw**) to create a "child" object like a surface, the child uses the **IUnknown::AddRef** method of the "parent" DirectDraw object to increment the parent's reference count.

When your application no longer needs an object, call the **Release** method to decrement its reference count. When the count reaches zero, the object is removed from memory. When a child object's reference count reaches zero, it calls the parent's **IUnknown::Release** method to indicate that there is one less object who will be needing the parent's services.

Implicitly allocated objects, such as the back-buffer surfaces in a flipping chain that you create with a single **IDirectDraw4::CreateSurface** call, are automatically deallocated when their parent DirectDrawSurface object is released. Also, you can only release a DirectDraw object from the thread that created the application window. For single-threaded applications, this restriction obviously doesn't apply, as there is only one thread. If your application created a primary flipping chain of two surfaces (created by a single **CreateSurface** call) that used an attached DirectDrawClipper object, the code to release these objects safely might look like:

```
// For this example, the g_lpDDraw, g_lpDDSurface, and
// g_lpDDClip are valid pointers to objects.
void ReleaseDDDrawObjects(void)
{
    // If the DirectDraw object pointer is valid,
    // it should be safe to release it and the objects it owns.
    if(g_lpDDraw)
    {
        // Release the DirectDraw object. (This call wouldn't
        // be safe if the children were created through IDirectDraw2
        // or IDirectDraw. See the following note for
        // more information)
        g_lpDDraw->Release(), g_lpDDraw = NULL;

        // Now, release the clipper that is attached to the surfaces.
        if(g_lpDDClip)
            g_lpDDClip->Release(), g_lpDDClip = NULL;

        // Now, release the primary flipping chain. Note
        // that this is only valid because the flipping
        // chain surfaces were created with a single
        // CreateSurface call. If they were explicitly
        // created and attached, then they must also be
        // explicitly released.
        if(g_lpDDSurface)
            g_lpDDSurface->Release(), g_lpDDSurface = NULL;
    }
}
```

Note

Earlier versions of the DirectDraw interface (**IDirectDraw2** and **IDirectDraw**, to be exact) behave differently than the most recent interface. When using these early interfaces, DirectDraw automatically releases all child objects when the parent itself is released. As a result, if you use these older interfaces, the order in which you release objects is critical. In this case, you should release the children of a DirectDraw object before releasing the DirectDraw object itself (or not release them at all,

counting on the parent to do cleanup for you). Because the `DirectDraw` object releases the child objects, if you release the parent before the children, you are very likely to incur a memory fault for attempting to dereference a pointer that was invalidated when the parent object released its children.

Some older applications relied on the automatic release of child objects and neglected to properly release some objects when no longer needed. At the time, this practice didn't cause any negative side effects, however doing so when using the `IDirectDraw4` interface might result in memory leaks.

Multiple DirectDraw Objects per Process

`DirectDraw` allows a process to call the `DirectDrawCreate` function as many times as necessary. A unique and independent interface to a unique and independent `DirectDraw` object is returned after each call. Each `DirectDraw` object can be used as desired; there are no dependencies between the objects. Each object behaves exactly as if it had been created by a unique process.

`DirectDraw` objects are independent of one another and the `DirectDrawSurface`, `DirectDrawPalette`, and `DirectDrawClipper` objects they create should not be used with other `DirectDraw` objects because they are automatically released when the parent `DirectDraw` object is destroyed. If they are used with another `DirectDraw` object, they might stop functioning if their parent object is destroyed, causing the remaining `DirectDraw` object to malfunction.

The exception is `DirectDrawClipper` objects created by using the `DirectDrawCreateClipper` function. These objects are independent of any particular `DirectDraw` object and can be used with one or more `DirectDraw` objects.

Creating DirectDraw Objects by Using CoCreateInstance

You can create a `DirectDraw` object by using the `CoCreateInstance` function and the `IDirectDraw4::Initialize` method rather than the `DirectDrawCreate` function. The following steps describe how to create the `DirectDraw` object:

- 1 Initialize COM at the start of your application by calling `CoInitialize` and specifying `NULL`.

```
if (FAILED(CoInitialize(NULL)))
    return FALSE;
```

- 2 Create the `DirectDraw` object by using `CoCreateInstance` and the `IDirectDraw4::Initialize` method.

```
ddrval = CoCreateInstance(&CLSID_DirectDraw,
    NULL, CLSCTX_ALL, &IID_IDirectDraw4, &lppd);
if (!FAILED(ddrval))
    ddrval = IDirectDraw4_Initialize(lppd, NULL);
```

In this call to `CoCreateInstance`, the first parameter, `CLSID_DirectDraw`, is the class identifier of the `DirectDraw` driver object class, the `IID_IDirectDraw4` parameter identifies the particular `DirectDraw` interface to be created, and the `lppd` parameter points to the `DirectDraw` object that is retrieved. If the call is successful, this function returns an uninitialized object.

- 3 Before you use the `DirectDraw` object, you must call `IDirectDraw4::Initialize`. This method takes the driver GUID parameter that the `DirectDrawCreate` function typically uses (`NULL` in this case). After the `DirectDraw` object is initialized, you can use and release it as if it had been created by using the `DirectDrawCreate` function. If you do not call the `Initialize` method before using one of the methods associated with the `DirectDraw` object, a `DDERR_NOTINITIALIZED` error will occur.

Before you close the application, close the COM library by using the `CoUninitialize` function.

```
CoUninitialize();
```

Surfaces

This section contains information about `DirectDrawSurface` objects. The following topics are discussed:

- Basic Concepts of Surfaces
- Creating Surfaces
- Flipping Surfaces
- Blitting to Surfaces
- Losing and Restoring Surfaces

- COM Reference Count Semantics for Surfaces
- Enumerating Surfaces
- Updating Surface Characteristics
- Accessing Surface Memory Directly
- Gamma and Color Controls
- Overlay Surfaces
- Compressed Texture Surfaces
- Private Surface Data
- Surface Uniqueness Values
- Using Non-local Video Memory Surfaces
- Converting Color and Format
- Surfaces and Device Contexts

Basic Concepts of Surfaces

This section contains information about the basic concepts associated with `DirectDrawSurface` objects. The following topics are discussed:

- What Are Surfaces?
- Surface Interfaces
- Width vs. Pitch
- Color Keying
- Pixel Formats

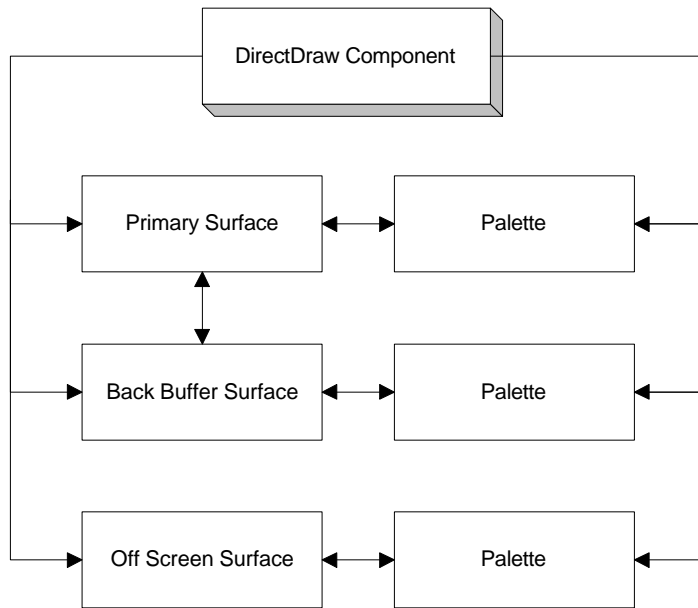
What Are Surfaces?

A surface, or `DirectDrawSurface` object, represents a linear area of display memory. A surface usually resides in the display memory of the display card, although surfaces can exist in system memory. Unless specifically instructed otherwise during the creation of the `DirectDrawSurface` object, `DirectDraw` object will put the `DirectDrawSurface` object wherever the best performance can be achieved given the requested capabilities. `DirectDrawSurface` objects can take advantage of specialized processors on display cards, not only to perform certain tasks faster, but to perform some tasks in parallel with the system CPU.

Using the `IDirectDraw4::CreateSurface` method, you can create a single surface object, complex surface-flipping chains, or three-dimensional surfaces. The `CreateSurface` method creates the requested surface or flipping chain and retrieves a pointer to the primary surface's `IDirectDrawSurface4` interface through which the object exposes its functionality.

The `IDirectDrawSurface4` interface enables you to indirectly access memory through blit methods, such as `IDirectDrawSurface4::BltFast`. The surface object can provide a device context to the display that you can use with GDI functions. Additionally, you can use `IDirectDrawSurface4` methods to directly access display memory. For example, you can use the `IDirectDrawSurface4::Lock` method to lock the display memory and retrieve the address corresponding to that surface. Addresses of display memory might point to visible frame buffer memory (primary surface) or to nonvisible buffers (off-screen or overlay surfaces). Nonvisible buffers usually reside in display memory, but can be created in system memory if required by hardware limitations or if `DirectDraw` is performing software emulation. In addition, the `IDirectDrawSurface4` interface extends other methods that you can use to set or retrieve palettes, or to work with specific types or surfaces, like flipping chains or overlays.

From this illustration, you can see that all surface are created by a `DirectDraw` object and are often used closely with palettes. Although each surface object can be assigned a palette, palettes aren't required for anything but primary surfaces that use pixel formats of 8-bits in depth or less.



Surface Interfaces

DirectDrawSurface objects expose their functionality through the **IDirectDrawSurface**, **IDirectDrawSurface2**, **IDirectDrawSurface3**, and **IDirectDrawSurface4** interfaces. Each new interface version provides the same utility as its predecessors, with additional options available through new methods.

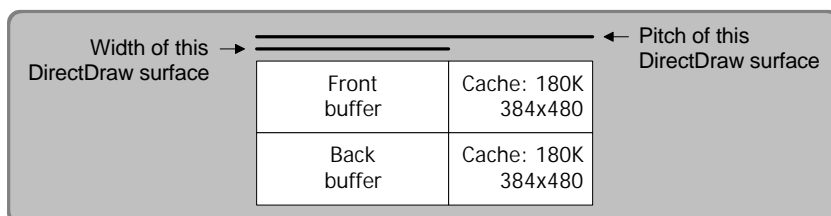
When you create a surface by calling the **IDirectDraw4::CreateSurface** method (or another creation method from **IDirectDraw4**), you receive a pointer to the surface's **IDirectDrawSurface4** interface. This behavior is different than previous versions of DirectX. Before the introduction of the **IDirectDraw4** interface, the **CreateSurface** method provided a pointer to a surface's **IDirectDrawSurface** interface. If you wanted to work with a different iteration of the interface, you had to query for it. When using **IDirectDraw4** this isn't the case, although you are free to query a surface for a previous iteration of an interface if you choose.

Width vs. Pitch

Although the terms *width* and *pitch* are discussed casually, they have very important (and distinctly different) meanings. As a result, you should understand the meanings for each, and how to interpret the values that DirectDraw uses to describe them.

DirectDraw uses the **DDSURFACEDESC2** structure to carry information describing a surface. Among other things, this structure is defined to contain information about a surface's dimensions, as well as how those dimensions are represented in memory. The structure uses the **dwHeight** and **dwWidth** members to describe the logical dimensions of the surface. Both of these members are measured in pixels. Therefore, the **dwHeight** and **dwWidth** values for a 640×480 surface are the same whether it is an 8-bit palettized surface or a 24-bit RGB surface.

The **DDSURFACEDESC2** structure contains information about how a surface is represented in memory through the **IPitch** member. The value in the **IPitch** member describes the surface's memory pitch (also called *stride*). Pitch is the distance, in bytes, between two memory addresses that represent the beginning of one bitmap line and the beginning of the next bitmap line. Because pitch is measured in bytes rather than pixels, a 640×480×8 surface will have a very different pitch value than a surface with the same dimensions but a different pixel format. Additionally, the pitch value sometimes reflects bytes that DirectDraw has reserved as a cache, so it is not safe to assume that pitch is simply the width multiplied by the number of bytes per pixel. Rather, you could visualize the difference between width and pitch as shown in the following illustration.



In this figure, the front buffer and back buffer are both 640×480×8, and the cache is 384×480×8.

Pitch values are useful when you are directly accessing surface memory. For example, after calling the **IDirectDrawSurface4::Lock** method, the **lpSurface** member of the associated **DDSURFACEDESC2** structure contains the address of the top-left pixel of the locked area of the surface, and the **IPitch** member is the surface pitch. You access pixels horizontally by incrementing or decrementing the surface pointer by the number of bytes per pixel, and you move up or down by adding the pitch value to, or subtracting it from, the current surface pointer.

When accessing surfaces directly, take care to stay within the memory allocated for the dimensions of the surface and stay out of any memory reserved for cache. Additionally, when you lock only a portion of a surface, you must stay within the rectangle you specify when locking the surface. Failing to follow these guidelines will have unpredictable results. When rendering directly into surface memory, always use the pitch returned by the **Lock** method (or the **IDirectDrawSurface4::GetDC** method). Do not assume a pitch based solely on the display mode. If your application works on some display adapters but looks garbled on others, this may be the cause of your problem.

For more information, see [Accessing Surface Memory Directly](#).

Color Keying

DirectDraw supports source and destination *color keying* for blits and overlay surfaces. Color keys enable you to display one image on top of another selectively, so that only certain pixels from the foreground rectangle are displayed, or only certain pixels on the background rectangle are overwritten.

You supply a single color key or a range of colors for source or destination color keying by calling the **IDirectDrawSurface4::SetColorKey** method.

For more information about color keying, see the following topics:

- [Overlay Color Keys](#)
- [Transparent Blitting](#)

Pixel Formats

Pixel formats dictate how data for each pixel in surface memory is to be interpreted. DirectDraw uses the **DDPIXELFORMAT** structure to describe various pixel formats. The **DDPIXELFORMAT** contains members to describe the following traits of a pixel format:

- Palettized or non-palettized pixel format
- If non-palettized, whether the pixel format is RGB or YUV
- Bit depth
- Bit masks for the pixel format's components

You can retrieve information about an existing surface's pixel format by calling the **IDirectDrawSurface4::GetPixelFormat** method.

Creating Surfaces

The **DirectDrawSurface** object represents a surface that usually resides in the display memory, but can exist in system memory if display memory is exhausted or if it is explicitly requested.

Use the **IDirectDraw4::CreateSurface** method to create one surface or to simultaneously create multiple surfaces (a complex surface). When calling **CreateSurface**, you specify the dimensions of the surface, whether it is a single surface or a complex surface, and the pixel format (if the surface won't be using an indexed palette). All these characteristics are contained in a **DDSURFACEDESC2** structure, whose address you send with the call. If the hardware can't support the requested capabilities or if it previously allocated those resources to another **DirectDrawSurface** object, the call will fail.

Creating single surfaces or multiple surfaces is a simple matter that requires only a few lines of code. There are a few common situations (and some less common ones) in which you will need to create surfaces. The following situations are discussed:

- [Creating the Primary Surface](#)
- [Creating an Off-Screen Surface](#)
- [Creating Complex Surfaces and Flipping Chains](#)
- [Creating Wide Surfaces](#)
- [Creating Client Memory Surfaces](#)

By default, for all surfaces except client memory surfaces, DirectDraw attempts to create a surface in local video memory. If there isn't enough local video memory available to hold the surface, DirectDraw will try to use non-local video memory (on some Accelerated Graphics Port-equipped systems), and fall back on system memory if all other types of memory are unavailable. You can explicitly request that a surface be created in a certain type of memory by including the appropriate flags in the associated **DDSCAPS2** structure when calling **IDirectDraw4::CreateSurface**.

Creating the Primary Surface

The primary surface is the surface currently visible on the monitor and is identified by the **DDSCAPS_PRIMARYSURFACE** flag. You can only have one primary surface for each DirectDraw object.

When you create a primary surface, remember that the dimensions and pixel format implicitly match the current display mode. Therefore, this is the one time you don't need to declare a surface's dimensions or pixel format. If you do specify them, the call will fail and return **DDERR_INVALIDPARAMS**—even if the information you used matches the current display mode.

The following example shows how to prepare the **DDSURFACEDESC2** structure members relevant for creating the primary surface.

```
DDSURFACEDESC2 ddsd;
ddsd.dwSize = sizeof(dds);

// Tell DirectDraw which members are valid.
ddsd.dwFlags = DDSD_CAPS;

// Request a primary surface.
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

After creating the primary surface, you can retrieve information about its dimensions and pixel format by calling its **IDirectDrawSurface4::GetSurfaceDesc** method.

See also, Display Modes.

Creating an Off-Screen Surface

An off-screen surface is often used to cache bitmaps that will later be blitted to the primary surface or a back buffer. You must declare the dimensions of an off-screen surface by including the **DDSD_WIDTH** and **DDSD_HEIGHT** flags and the corresponding values in the **dwWidth** and **dwHeight** members. Additionally, you must include the **DDSCAPS_OFFSCREENPLAIN** flag in the accompanying **DDSCAPS2** structure.

By default, DirectDraw creates a surface in display memory unless it will not fit, in which case it creates the surface in system memory. You can explicitly choose display or system memory by including the **DDSCAPS_SYSTEMMEMORY** or **DDSCAPS_VIDEOMEMORY** flags in the **dwCaps** member of the **DDSCAPS2** structure. The method fails, returning an error, if it can't create the surface in the specified location.

The following example shows how to prepare for creating a simple off-screen surface:

```
DDSURFACEDESC2 ddsd;
ddsd.dwSize = sizeof(dds);

// Tell DirectDraw which members are valid.
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;

// Request a simple off-screen surface, sized
// 100 by 100 pixels.
//
// (This assumes that the off-screen surface we are about
// to create will match the pixel format of the
// primary surface.)
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 100;
ddsd.dwWidth = 100;
```

Additionally, you can create surfaces whose pixel format differs from the primary surface's pixel format. However, in this case there is one drawback—you are limited to using system memory. The following code fragment shows how to prepare the **DDSURFACEDESC2** structure members in order to create an 8-bit palettized surface (assuming that the current display mode is something other than 8-bits per pixel).

```
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDS_DCAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_SYSTEMMEMORY;
ddsd.dwHeight = 100;
ddsd.dwWidth = 100;
ddsd.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
ddsd.ddpfPixelFormat.dwFlags = DDPF_RGB | DDPF_PALETTEINDEXED8;

// Set the bit depth for an 8-bit surface, but DO NOT
// specify any RGB mask values. The masks must be zero
// for a palettized surface.
ddsd.ddpfPixelFormat.dwRGBBitCount = 8;
```

In previous versions of DirectX, the maximum width of off-screen surfaces was limited to the width of the primary surface. Beginning with DirectX 5.0, you can create surfaces as wide as you need, permitting that the display hardware can support them. Be careful when declaring wide off-screen surfaces; if the video card memory cannot hold a surface as wide as you request, the surface is created in system memory. If you explicitly choose video memory and the hardware can't support it, the call fails. For more information, see [Creating Wide Surfaces](#).

Creating Complex Surfaces and Flipping Chains

You can also create *complex surfaces*. A complex surface is a set of surfaces created with a single call to the **IDirectDraw4::CreateSurface** method. If the **DDSCAPS_COMPLEX** flag is set when you call **CreateSurface** call, DirectDraw implicitly creates one or more surfaces in addition to the surface explicitly specified. You manage complex surfaces just like a single surface—a single call to the **IDirectDraw::Release** method releases all surfaces, and a single call to the **IDirectDrawSurface4::Restore** method restores them all. However, implicitly created surfaces cannot be detached. For more information, see **IDirectDrawSurface4::DeleteAttachedSurface**.

One of the most useful complex surfaces you can create is a flipping chain. Usually, a flipping chain is made of a primary surface and one or more back buffers. The **DDSCAPS_FLIP** flag indicates that a surface is part of a flipping chain. Creating a flipping chain this way requires that you also include the **DDSCAPS_COMPLEX** flag.

The following example shows how to prepare for creating a primary surface flipping chain.

```
DDSURFACEDESC2 ddsd2;
ddsd2.dwSize = sizeof(ddsd2);

// Tell DirectDraw which members are valid.
ddsd2.dwFlags = DDS_DCAPS | DDSD_BACKBUFFERCOUNT;

// Request a primary surface with a single
// back buffer
ddsd2.ddsCaps.dwCaps = DDSCAPS_COMPLEX | DDSCAPS_FLIP |
DDSCAPS_PRIMARYSURFACE;
ddsd2.dwBackBufferCount = 1;
```

The previous example constructs a double-buffered flipping environment—a single call to the **IDirectDrawSurface4::Flip** method exchanges the surface memory of the primary surface and the back buffer. If you specify 2 for the value of the **dwBackBufferCount** member of the **DDSURFACEDESC2** structure, two back buffers are created, and each call to **Flip** rotates the surfaces in a circular pattern, providing a triple-buffered flipping environment. For more information, see [Flipping Surfaces](#).

Note

To create a flipping chain that comprises surfaces that will be used as 3-D render targets, be sure to include the **DDSCAPS_3DDEVICE** capability flag in the surface description, as well as the **DDSCAPS_COMPLEX** and **DDSCAPS_FLIP** flags.

Unlike the **CreateSurface** method exposed by the **IDirectDraw3** and earlier interfaces, you cannot use **IDirectDraw4::CreateSurface** to implicitly create a flipping chain of render target surfaces with an attached depth-buffer. The **DDSURFACEDESC2** structure that the **IDirectDraw4::CreateSurface** method accepts doesn't contain a field to specify a depth-buffer bit depth. As a result, applications must create a depth-buffer surface explicitly, then attach it to the back-buffer render target surface. For more information, see [Depth Buffers](#).

Creating Wide Surfaces

DirectDraw allows you to create off-screen surfaces in video memory that are wider than the primary surface. This is only possible when display device support for wide surfaces is present.

To check for wide surface support, call **IDirectDraw4::GetCaps** and look for the **DDCAPS2_WIDESURFACES** flag in the **dwCaps2** member of the first **DDCAPS** structure you send with the call. If the flag is present, you can create video memory off-screen surfaces that are wider than the primary surface.

If you attempt to create a wide surface in video memory when the **DDCAPS2_WIDESURFACES** flag isn't present, the attempt will fail and return **DDERR_INVALIDPARAMS**. Note that attempting to create extremely large surfaces might still fail, even if the driver exposes the **DDCAPS2_WIDESURFACES** flag.

Wide surfaces are always supported for system memory surfaces, video port surfaces, and execute buffers.

Creating Client Memory Surfaces

Client memory surfaces are simply **DirectDrawSurface** objects that use system memory that your application has previously allocated to hold image data. Creating such a surface isn't common, but it isn't difficult to do and it can be useful for applications that need to use **DirectDraw** surface features on existing memory buffers.

Like creating all surfaces, **DirectDraw** needs information about the dimensions of the surface (measured in pixels) and the surface pitch (measured in bytes), as well as the surface's pixel format. However, unlike creating other types of surfaces, this information doesn't tell **DirectDraw** how you want the surface to be created, it tells **DirectDraw** how you've already created it. You set these characteristics, plus the memory address of the buffer you've allocated, in the **DDSURFACEDESC2** structure you pass to the **IDirectDraw4::CreateSurface** method.

A client memory surface works just like a normal system-memory surface, with the exception that **DirectDraw** does not attempt to free the surface memory when it's no longer needed; freeing client allocated memory is the application's responsibility.

The following example shows how you might allocate memory and create a **DirectDrawSurface** object for a 64×64 pixel 24-bit RGB surface:

```
// For this example, g_lpDD4 is a valid IDirectDraw4 interface pointer.

#define WIDTH 64 // in pixels
#define HEIGHT 64
#define DEPTH 3 // in bytes (3bytes == 24 bits)

HRESULT hr;
LPVOID lpSurface = NULL;
HLOCAL hMemHandle = NULL;
DDSURFACEDESC2 ddsd2;
LPDIRECTDRAWSURFACE4 lpDDS4;

// Allocate memory for a 64 by 64, 24-bit per pixel buffer.
// REMEMBER: The application is responsible for freeing this
//           buffer when it is no longer needed.
if (lpSurface = malloc((size_t)WIDTH*HEIGHT*DEPTH))
    ZeroMemory(lpSurface, (DWORD)WIDTH*HEIGHT*DEPTH);
else
    return DDERR_OUTOFMEMORY;

// Initialize the surface description.
ZeroMemory(&ddsd2, sizeof(DDSURFACEDESC2));
ZeroMemory(&ddsd2.ddpfPixelFormat, sizeof(DDPIXELFORMAT));
ddsd2.dwSize = sizeof(ddsd2);
```

```

ddsd2.dwFlags = DDSD_WIDTH | DDSD_HEIGHT | DDSD_LPSURFACE |
                DDSD_PITCH | DDSD_PIXELFORMAT;
ddsd2.dwWidth = WIDTH;
ddsd2.dwHeight = HEIGHT;
ddsd2.lPitch = (LONG)DEPTH * WIDTH;
ddsd2.lpSurface = lpSurface;

// Set up the pixel format for 24-bit RGB (8-8-8).
ddsd2.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
ddsd2.ddpfPixelFormat.dwFlags = DDPF_RGB;
ddsd2.ddpfPixelFormat.dwRGBBitCount = (DWORD)DEPTH*8;
ddsd2.ddpfPixelFormat.dwRBitMask = 0x00FF0000;
ddsd2.ddpfPixelFormat.dwGBitMask = 0x0000FF00;
ddsd2.ddpfPixelFormat.dwBBitMask = 0x000000FF;

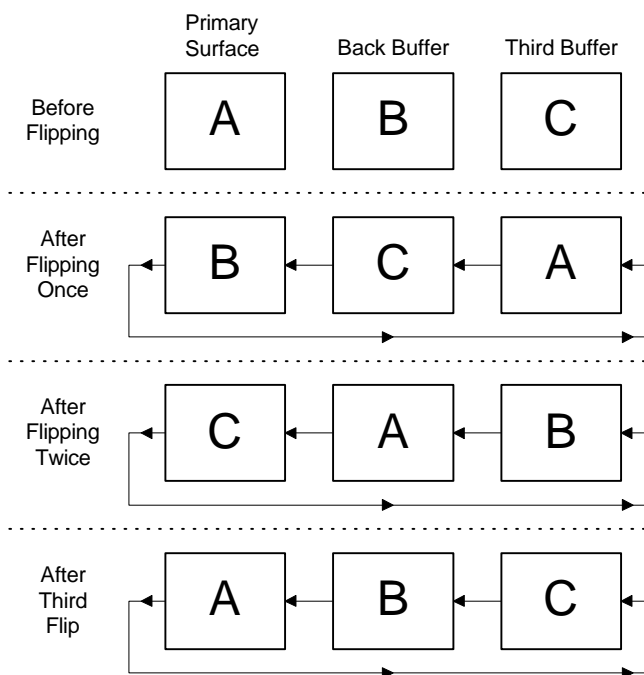
// Create the surface
hr = g_lpDD4->CreateSurface(&ddsd2, &lpDDS4, NULL);
return hr;

```

Flipping Surfaces

Any surface in DirectDraw can be constructed as a *flipping surface*. A flipping surface is any piece of memory that can be swapped between a *front buffer* and a *back buffer*. (This construct is commonly referred to as a *flipping chain*). Often, the front buffer is the primary surface, but it doesn't have to be.

Typically, when you use the **IDirectDrawSurface4::Flip** method to request a surface flip operation, the pointers to surface memory for the primary surface and back buffers are swapped. Flipping is performed by switching pointers that the display device uses for referencing memory, not by copying surface memory. (The exception to this is when DirectDraw is emulating the flip, in which case it simply copies the surfaces. DirectDraw emulates flip operations if a back buffer cannot fit into display memory or if the hardware doesn't support DirectDraw.) When a flipping chain contains a primary surface and more than one back buffer, the pointers are switched in a circular pattern, as shown in the following illustration.



Other surfaces that are attached to a DirectDraw object, but not part of the flipping chain, are unaffected when the **Flip** method is called.

Remember, DirectDraw flips surfaces by swapping surface memory pointers within DirectDrawSurface objects, not by swapping the objects themselves. This means that, to blit to the back buffer in any type of flipping scheme, you always use the same

DirectDrawSurface object—the one that was the back buffer when you created the flipping chain. Conversely, you always perform a flip operation by calling the front surface's **Flip** method.

When working with visible surfaces, such as a primary surface flipping chain or a visible overlay surface flipping chain, the **Flip** method is asynchronous unless you include the DDFLIP_WAIT flag. On these visible surfaces, the **Flip** method can return before the actual flip operation occurs in the hardware (because the hardware doesn't flip until the next vertical refresh occurs). While the actual flip operation is pending, the back buffer behind the currently visible surface can't be locked or blitted by calling the **IDirectDrawSurface4::Lock**, **IDirectDrawSurface4::Blt**, **IDirectDrawSurface4::BltFast**, or **IDirectDrawSurface4::GetDC** methods. If you attempt to call these methods while a flip operation is pending, they will fail and return DDERR_WASSTILLDRAWING. However, if you are using a triple buffered scheme, the rearmost buffer is still available.

Blitting to Surfaces

This section is a guide to copying pixels from one DirectDraw surface to another, or from one part of a surface to another.

The following topics are covered:

- Blitting Basics
- Blitting with BltFast
- Blitting with Blt
- Blit Timing
- Transparent Blitting
- Color Fills
- Blitting to Multiple Windows

Blitting Basics

Two methods are available for copying images to a DirectDraw surface: **IDirectDrawSurface4::Blt** and **IDirectDrawSurface4::BltFast**. (A third method, **IDirectDrawSurface4::BltBatch**, is not implemented in this version of DirectX.) These methods are called on the destination surface and receive the source surface as a parameter. The destination and source surfaces can be one and the same, and you don't have to worry about overlap—DirectDraw takes care to preserve all source pixels before overwriting them.

Of the two implemented methods, **Blt** is the more flexible and **BltFast** is the faster—but only if there is no hardware blitter. You can determine the blitting capabilities of the hardware from the **DDCAPS** structure obtained in the *lpDDDriverCaps* parameter of the **IDirectDraw4::GetCaps** method. If the **dwCaps** member contains DDCAPS_BLT, the hardware has at least minimal blitting capabilities.

Blitting with BltFast

When using **IDirectDrawSurface4::BltFast**, you supply a valid rectangle in the source surface from which the pixels are to be copied (or NULL to specify the entire surface), and an x-coordinate and y-coordinate in the destination surface. The source rectangle must be able to fit in the destination surface with its top left corner at that point, or the call will fail with a return value of DDERR_INVALIDRECT. **BltFast** cannot be used on surfaces that have an attached clipper.

No stretching, mirroring, or other effects can be performed when using **BltFast**.

BltFast Example

The following example copies pixels from an offscreen surface, *lpDDSOffOne*, to the primary surface, *lpDDSPrimary*. The flags ensure that the operation will take place as soon as the blitter is free, and that transparent pixels in the source image will not be copied. (For more information on the meaning of these flags, see Blit Timing and Transparent Blitting .)

```
lpDDSPrimary->BltFast(
    100, 200,    // Upper left xy of destination
    lpDDSOffOne, // Source surface
    NULL,       // Source rectangle = entire surface
    DDBLTFAST_WAIT | DDBLTFAST_SRCCOLORKEY );
```

Blitting with Blt

When using the **IDirectDrawSurface4::Blt** method, you supply a valid rectangle in the source surface (or NULL to specify the entire surface), and a rectangle in the destination surface to which the source image will be copied (again, NULL means the rectangle covers the entire surface). If a clipper is attached to the destination surface, the bounds of the destination rectangle can fall outside the surface and clipping will be performed. If there is no clipper, the destination rectangle must fall entirely within the surface or else the method will fail with DDERR_INVALIDRECT. (For more information on clipping, see Clippers.)

Scaling

The **Blt** method automatically rescales the source image to fit the destination rectangle. If resizing is not your intention, for best performance you should make sure that your source and destination rectangles are exactly the same size, or else use **IDirectDrawSurface4::BltFast**. (See Blitting with BltFast.)

Hardware acceleration for scaling depends on the DDFXCAPS_BLT* flags in the **dwFXCaps** member of the **DDCAPS** structure for the device. If, for example, a device has the DDFXCAPS_BLTSTRETCHXN capability but not DDFXCAPS_BLTSTRETCHX, it can assist when the x-axis of the source rectangle is being multiplied by a whole number but not when non-integral (arbitrary) scaling is being done.

Devices might also support arithmetic scaling, which is scaling by interpolation rather than simple multiplication or deletion of pixels. For instance, if an axis was being increased by one-third, the pixels would be recolored to provide a closer approximation to the original image than would be produced by the doubling of every third pixel on that axis.

Applications cannot control the type of scaling done by the driver, except by setting the DDBLTFX_ARITHSTRETCHY flag in the **dwDDFX** member of the **DDBLTFX** structure passed to **Blt**. This flag requests that arithmetic stretching be done on the y-axis. Arithmetic stretching on the x-axis and arithmetic shrinking are not currently supported in the DirectDraw API, but a driver may perform them by default.

Other Effects

If you do not require any special effects other than scaling when using **Blt**, you can pass NULL as the *lpDDBlitFx* parameter. Otherwise you can choose among a variety of effects specified in a **DDBLTFX** structure. Among these, color fills and mirroring are supported by the HEL, so they are always available. Most other effects depend on hardware support.

For a complete view of the effects capabilities of the HEL, run the DDraw Caps utility supplied with the DirectX Programmer's Reference and select HEL FX Caps from the HEL menu. For an explanation of the various flags, see **DDCAPS**. You can also check HEL capabilities within your own application by using the **IDirectDraw4::GetCaps** method.

When you specify an effect that requires a value in one of the members of the **DDBLTFX** structure passed to the **IDirectDrawSurface4::Blt** method, you must also include the appropriate flags in the *dwFlags* parameter to show which members of the structure are valid.

Some effects require only the setting of a flag in the **dwFlags** member of **DDBLTFX**. One of these is DDBLTFX_NOTEARING. You can use this flag when you are blitting animated images directly to the front buffer, so that the blit is timed to coincide with the screen refresh and the possibility of tearing is minimized. Mirroring and rotation are also set by using flags.

Blitting effects include the standard raster operations (ROPs) used by GDI functions such as **BitBlt**. The only ROPs supported by the HEL are SRCCOPY (the default), BLACKNESS, and WHITENESS. Hardware support for other ROPs can be examined in the **DDCAPS** structure for the driver. If you wish to use any of the standard ROPS with the **Blt** method, you flag them in the **dwROP** member of the **DDBLTFX** structure.

The **dwDDROP** member of the **DDBLTFX** structure is for specifying ROPs specific to DirectDraw. However, no such ROPs are currently defined.

Alpha and Z Values

Opacity and depth values are not currently supported in DirectDraw blits. If alpha values are stored in the pixel format, they simply overwrite any alpha values in the destination rectangle. Values from alpha buffers and z-buffers are ignored. The members of the **DDBLTFX** structure that have to do with alpha channels and z-buffers (members whose names begin with "dwAlpha" and "dwZ"), and the corresponding flags for **Blt**, are not used. The same applies to the DDBLTFX_ZBUFFERBASEDEST and DDBLTFX_ZBUFFERRANGE flags in the **dwDDFX** member of the **DDBLTFX** structure.

Although z-buffers are currently used only in Direct3D applications, you can use **IDirectDrawSurface4::Blt** to set the depth value for a z-buffer surface, by setting the DDBLT_DEPTHFILL flag. For more information, see Clearing Depth Buffers.

For an overview of the use of alpha channels and z-buffers in Direct3D, see the following topics:

- Alpha States

- What Are Depth Buffers?

Blt Example

The following example, in which it is assumed that *lpDDS* is a valid **IDirectDrawSurface4** pointer, creates a symmetrical image within the surface by mirroring a rectangle from left to right:

```
RECT    rcSource, rcDest;
DDBLTFX ddbl tfx;

ZeroMemory(&ddb l tfx, sizeof(ddbl tfx));
ddb l tfx.dwSize = sizeof(ddbl tfx);
ddb l tfx.dwDDFX = DDBLTFX_MIRRORLEFTRIGHT;

rcSource.top = 0; rcSource.left = 0;
rcSource.bottom = 100; rcSource.right = 200;
rcDest.top = 0; rcDest.left = 201;
rcDest.bottom = 100; rcDest.right = 401;

HRESULT hr = lpDDS->Bl t(&rcDest,
                        lpDDS,
                        &rcSource,
                        DDBLT_WAIT | DDBLT_DDFX,
                        &ddb l tfx);
```

Blit Timing

When you copy pixels to a surface using either **IDirectDrawSurface4::Blt** or **IDirectDrawSurface4::BltFast**, the method might fail with `DDERR_WASSTILLDRAWING` because the hardware blitter was not ready to accept the command.

If your application has no urgent business to perform while waiting for the blitter to come back into a state of readiness, you can specify the `DDBLT_WAIT` flag in the *dwFlags* parameter of **Blt**, or the equivalent `DDBLTFAST_WAIT` flag for **BltFast**. The flag causes the method to wait until the blit can be handed off to the blitter (or until an error other than `DDERR_WASSTILLDRAWING` occurs).

Blt accepts another flag, `DDBLT_ASYNC`, that takes advantage of any hardware FIFO (first in, first out) queuing capabilities.

Transparent Blitting

This section discusses the theory and practice of using transparent blitting to copy parts of a rectangular image selectively, using source and destination color keys.

The concepts are introduced in the following topic:

- What Is Transparent Blitting?

Information about the implementation of transparent blitting in `DirectDraw` is contained in the following topics:

- Color Key Format
- Setting Color Keys
- Blitting with Color Keys

What Is Transparent Blitting?

Transparent blitting enables you to create the illusion of nonrectangular blits when animating sprites. A sprite image is usually nonrectangular, but blits are always rectangular, so every pixel within the sprite's bounding rectangle becomes part of the data transfer. With transparent blitting, each pixel that is not part of the sprite image is treated as transparent when the blitter is moving the image to its destination, so that it does not overwrite the color in that pixel on the background image.

The artist creating the sprite chooses an arbitrary color or range of colors to be used as the transparency color key. This is typically a single uncommon color that the artist doesn't use for anything but transparency, and it is used to fill in all parts of the sprite rectangle that are not part of the desired image. At run time you set the color key for the surface containing the sprite. (If you wish,

you can automatically set it to the color of the pixel in the upper left corner of the image.) Subsequent blits can take advantage of that color key, ignoring the pixels that match it. This type of color key is known as a source color key.

You can also use a color key on the destination surface, provided the hardware supports destination color keying. This destination color key is used for pixels that can be overwritten by a sprite. For example, the artist might be working on a foreground image that sprites are supposed to pass behind, such as the wall of a room with a window to the outside. The artist chooses an arbitrary color—one that isn't used elsewhere in the image—to represent the sky outside the window. When you set this color key for the destination surface and then blit a sprite to that surface, the sprite's pixels will overwrite only pixels that are using the destination color key. In the example, the sprite appears only in the window, but not on the wall or window frame. As a result, the sprite seems to be outside the room.

Source and destination color keys can be combined. In the example, the sprite could use a source color key so that its entire bounding rectangle does not block out the sky background.

Color Key Format

A color key is described in a **DDCOLORKEY** structure. If the color key is a single color, both members of this structure should be assigned the same value. Otherwise the color key is a range of colors.

Color keys are specified using the pixel format of a surface. If a surface is in a palettized format, the color key is given as an index or a range of indices. If the surface's pixel format is specified by a FOURCC code that describes a YUV format, the YUV color key is specified by the three low-order bytes in both the **dwColorSpaceLowValue** and **dwColorSpaceHighValue** members of the **DDCOLORKEY** structure. The lowest order byte contains the V data, the second lowest order byte contains the U data, and the highest order byte contains the Y data.

Some examples of valid color keys follow:

8-bit palettized mode

```
// Palette entry 26 is the color key.
dwColorSpaceLowValue = 26;
dwColorSpaceHighValue = 26;
```

24-bit true-color mode

```
// Color 255, 128, 128 is the color key.
dwColorSpaceLowValue = RGBQUAD(255, 128, 128);
dwColorSpaceHighValue = RGBQUAD(255, 128, 128);
```

FourCC YUV mode

```
// Any YUV color where Y is between 100 and 110
// and U or V is between 50 and 55 is transparent.
dwColorSpaceLowValue = YUVQUAD(100, 50, 50);
dwColorSpaceHighValue = YUVQUAD(110, 55, 55);
```

Support for a range of colors rather than a single color is hardware-dependent. Check the **dwCKKeyCaps** member of the **DDCAPS** structure for the hardware. The HEL does not support color ranges.

Some hardware supports color ranges only for YUV pixel data, which is usually video. The transparent background in video footage (the "blue screen" against which the subject was photographed) might not be a single pure color, so a range of colors in the color key is desirable.

Setting Color Keys

You can set the source or destination color key for a surface either when creating it or afterwards.

To set a color key or keys when creating a surface, you assign the appropriate color values to one or both of the **ddckCKSrcBlit** and **ddckCKDestBlit** members of the **DDSURFACEDESC2** structure that is passed to **IDirectDraw4::CreateSurface**. To enable the color key for blitting, you must also include one or both of **DDSD_CKSRCLT** or **DDSD_CKDESTBLT** in the **dwFlags** member.

To set a color key for an existing surface you use the **IDirectDrawSurface4::SetColorKey** method. You specify a key in the *lpDDColorKey* parameter and set either **DDCKEY_SRCBLT** or **DDCKEY_DESTBLT** in the *dwFlags* parameter to indicated whether you are setting a source or destination key. If the **DDCOLORKEY** structure contains a range of colors, you must also set the **DDCKEY_COLORSPACE** flag. If this flag is not set, only the **dwColorSpaceLowValue** member of the structure is used.

Blitting with Color Keys

If you want to use color keys for surfaces when calling the **IDirectDrawSurface4::BltFast** method, you must set one or both of the **DDBLTFAST_SRCCOLORKEY** or **DDBLTFAST_DESTCOLORKEY** flags in the *dwTrans* parameter.

In order to use colors keys when calling **IDirectDrawSurface4::Blt**, you pass one or both of the **DDBLT_KEYSRC** or **DDBLT_KEYDEST** flags in the *dwFlags* parameter. Alternatively, you can put the appropriate color values in the **ddckDestColorkey** and **ddckSrcColorkey** members of the **DDBLTFX** structure that is passed to the method through the *lpDDBltFx* parameter. In this case you must also set the **DBLT_KEYSRCOVERRIDE** or **DDBLT_KEYDESTOVERRIDE** flag, or both, in the *dwFlags* parameter, so that the selected keys are taken from the **DDBLTFX** structure rather than from the surface properties.

Color Fills

In order to fill all or part of a surface with a single color, you can use the **IDirectDrawSurface4::Blt** method with the **DDBLT_COLORFILL** flag. This technique allows you to quickly erase an area or draw a solid-colored background.

The following example fills an entire surface with the color blue, after obtaining the numerical value for blue from the pixel format:

```

/* It is assumed that lpDDS is a valid pointer to
   an IDirectDrawSurface4 interface. */

HRESULT ddrval;
DDPIXELFORMAT ddpf;

ddpf.dwSize = sizeof(ddpf);
if (SUCCEEDED(lpDDS->GetPixelFormat(&ddpf))
{
    DDBLTFX ddbl tfx;

    ddbl tfx.dwSize = sizeof(ddbl tfx);
    ddbl tfx.dwFillColor = ddpf.dwBBitMask; // Pure blue

    ddrval = lpDDS->Blt(
        NULL,          // Destination is entire surface
        NULL,          // No source surface
        NULL,          // No source rectangle
        DDBLT_COLORFILL, &ddbl tfx);

    switch(ddrval)
    {
        case DDERR_WASSTILLDRAWING:
            .
            .
            .
        case DDERR_SURFACELOST:
            .
            .
            .
        case DD_OK:
            .
            .
            .
        default:
    }
}

```

Blitting to Multiple Windows

You can use a `DirectDraw` object and a `DirectDrawClipper` object to blit to multiple windows created by an application running at the normal cooperative level. For more information, see [Using a Clipper with Multiple Windows](#).

Creating multiple `DirectDraw` objects that blit to each others' primary surface is not recommended.

Losing and Restoring Surfaces

The surface memory associated with a `DirectDrawSurface` object may be freed, while the `DirectDrawSurface` objects representing these pieces of surface memory are not necessarily released. When a `DirectDrawSurface` object loses its surface memory, many methods return `DDERR_SURFACELOST` and perform no other action.

Surfaces can be lost because the display mode was changed or because another application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. The **`IDirectDrawSurface4::Restore`** method re-creates these lost surfaces and reconnects them to their `DirectDrawSurface` object. If your application uses more than one surface, you can call the **`IDirectDraw4::RestoreAllSurfaces`** method to restore all of your surfaces at once.

Restoring a surface doesn't reload any bitmaps that may have existed in the surface prior to being lost. You must completely reconstitute the graphics they once held.

COM Reference Count Semantics for Surfaces

Being built upon COM means that `DirectDraw` follows certain rules that employ reference counts to manage object lifetimes. For a conceptual overview, see the COM documentation; a `DirectDraw`-centered discussion of the topic is found in [Parent and Child Object Lifetimes](#).

By COM rules, when an interface pointer is copied by setting it to another variable or passing to another object, that copy represents another reference to the object, and therefore the **`IUnknown::AddRef`** method of the interface must be called to reflect the change. Not only should you follow COM reference counting rules when working with `DirectDraw` objects, but you should become familiar with the situations in which `DirectDraw` internally updates reference counts. Some `DirectDraw` methods—mostly those involving complex surface flipping chains—affect the reference counts of the surfaces involved, while methods involving clippers or palettes affect the reference counts of those objects. Knowing about these situations can make the difference in your application's stability and can prevent memory leaks. This section presents information divided into the following topics:

- When Reference Counts Will Change
- Reference Counts for Complex Surfaces
- Releasing Surfaces

Note:

There are some things to remember about the reference count of the `DirectDraw` object, in addition to the relationships discussed in this section. For more information, see [Parent and Child Object Lifetimes in The DirectDraw Object](#).

When Reference Counts will Change

There are several `DirectDraw` methods that affect the reference count of a surface, and a few that affect other objects you can associate with a surface. You can think of these situations as "surface-only changes" and "cross-object changes":

Surface-only changes

Surface-only changes, as the name states, only affect the reference count of a surface object. For example, you might use the **`IDirectDraw4::EnumSurfaces`** to enumerate the current surfaces that fit a particular description. When the method invokes the callback function that you provide, it passes a pointer to an **`IDirectDrawSurface4`** interface, but it increments the reference count for the object before your application receives the pointer. It's your responsibility to release the object when you are finished with it. This will typically be at the end of your callback routine, or later if you choose to keep the object.

Most other surface-only changes affect the reference counts of complex surfaces, such as a flipping chain. Reference counts are a little more tricky for complex surfaces, because (in most cases) `DirectDraw` treats a complex surface as if it was a single object, even though it is a set of surfaces. In short, the **`IDirectDrawSurface4::GetAttachedSurface`** and **`IDirectDrawSurface4::AddAttachedSurface`** methods increment reference counts of surfaces, and **`IDirectDrawSurface4::DeleteAttachedSurface`** decrements the reference count. These methods don't affect the counts of any surfaces attached to the current surface. See the references for these methods and [Reference Counts for Complex Surfaces](#) for additional details.

Cross-object changes

Cross-object reference count changes occur when you create an association between a surface and another object that performs a task for the surface, such as a clipper or a palette.

The **IDirectDrawSurface4::SetClipper** and **IDirectDrawSurface4::SetPalette** methods increment the reference count of the object being attached. After they are attached, the surface manages them; if the surface is released, it automatically releases any objects it is using. (For this reason, some applications release the interface for the object after these calls succeed. This is a perfectly valid practice.)

Once a clipper or palette is attached to a surface, you can call the **IDirectDrawSurface4::GetClipper** and **IDirectDrawSurface4::GetPalette** methods to retrieve them again. Because these methods return a copy of an interface pointer, they implicitly increment the reference count for the object being retrieved. When you're done with the interfaces, don't forget to release them—the objects that the interfaces represent won't disappear so long as the surface they are attached to still holds a reference to them.

Reference Counts for Complex Surfaces

The methods you use to manipulate a complex surface like a flipping chain all use surface interface pointers, and therefore they all affect the reference counts of the surfaces. Because a complex surface is really a series of single surfaces, the reference count relationships require a little more consideration. As you might expect, the **IDirectDrawSurface4::GetAttachedSurface** method returns the surface interface for a surface attached to the current surface. It does this after incrementing the reference count of the interface being retrieved; it's up to you to release the interface when you no longer need it. The

IDirectDrawSurface4::AddAttachedSurface method attaches a new surface to the current one. Similarly, **AddAttachedSurface** increments the count for the surface being attached. You would use the **IDirectDrawSurface4::DeleteAttachedSurface** method to remove the surface from the chain and implicitly decrease its reference count.

What isn't immediately clear about these methods is that they don't affect the reference counts of the other objects that make up the complex surface. The **GetAttachedSurface** method simply increments the reference count of the surface it's retrieving, it doesn't affect the counts of the surfaces on which it depends. (The same situation applies to an explicit call to **IUnknown::AddRef**.) This means that the reference count for primary surface in a complex surface can reach zero before its subordinate surfaces reach zero. When the primary surface reference count reaches zero, all other surfaces attached to it are released regardless of their current reference counts. (It's like a tree: if you cut the base, the whole thing falls. In this case, the primary surface is the base.) Attempts to access subordinate surfaces after the primary surface has been deallocated will result in memory faults.

To avoid problems, make sure that your application has released all subordinate surface references before attempting to release the primary surface. It might be helpful to track the references your application holds, only accessing subordinate surface interfaces when you're sure that you also hold a reference the primary surface.

Releasing Surfaces

Like all COM interfaces, you must release surfaces by calling their **IDirectDrawSurface4::Release** method when you no longer need them.

Each surface you individually create must be explicitly released. However, if you implicitly created multiple surfaces with a single call to **IDirectDraw4::CreateSurface**, such as a flipping chain, you need only release the front buffer. In this case, any pointers you might have to back buffer surfaces are implicitly released and can no longer be used.

Explicitly releasing a back buffer surface doesn't affect the reference count of the other surfaces in the chain.

Enumerating Surfaces

By calling the **IDirectDraw4::EnumSurfaces** method you can request that DirectDraw enumerate surfaces in various ways. The **EnumSurfaces** method enables you to look for surfaces that fit, or don't fit, a provided surface description. DirectDraw calls a **EnumSurfacesCallback** that you include with the call for each enumerated surface.

There are two general ways to search—you can search for surfaces that the DirectDraw object has already created, or for surfaces that the DirectDraw object is capable of creating at the time (given the surface description and available memory). You specify what type of search you want by combining flags in the method's *dwFlags* parameter.

Enumerating existing surfaces

This is the most common type of enumeration. You enumerate existing surfaces by calling **EnumSurfaces**, specifying a combination of the **DDENUMSURFACES_DOESEXIST** search-type flag and one of the matching flags (**DDENUMSURFACES_MATCH**, **DDENUMSURFACES_NOMATCH**, or **DDENUMSURFACES_ALL**) in the *dwFlags* parameter. If you're enumerating all existing surfaces, you can set the *lpDDSD* parameter to **NULL**, otherwise set it to the address

of an initialized **DDSURFACEDESC2** structure that describes the surface for which you're looking. You can set the third parameter, *lpContext*, to an address that will be passed to the enumeration function you specify in the fourth parameter, *lpEnumSurfacesCallback*.

The following code fragment shows what this call might look like to enumerate all of a DirectDraw object's existing surfaces.

```
HRESULT ddrval;
ddrval = lpDD->EnumSurfaces(DDENUMSURFACES_DOESEXIST |
                           DDENUMSURFACES_ALL, NULL, NULL,
                           EnumCallback);

if (FAILED(ddrval))
    return FALSE;
```

When searching for existing surfaces that fit a specific description, DirectDraw determines a match by comparing each member of the provided surface description to those of the existing surfaces. Only exact matches are enumerated. DirectDraw increments the reference counts of the enumerated surfaces, so make sure to release a surface if you don't plan to use it (or when you're done with it).

Enumerating possible surfaces

This type of enumeration is less common than enumerating existing surfaces, but it can be helpful to determine if a surface is supported before you attempt to create it. To perform this search, combine the **DDENUMSURFACES_CANBECREATED** and **DDENUMSURFACES_MATCH** flags when you call **IDirectDraw4::EnumSurfaces** (no other flag combinations are valid). The **DDSURFACEDESC2** structure you use with the call must be initialized to contain information about the surface characteristics that DirectDraw will use.

To enumerate surfaces that use a particular pixel format, include the **DDSD_PIXELFORMAT** flag in the **dwFlags** member of the **DDSURFACEDESC2** structure. Additionally, initialize the **DDPIXELFORMAT** structure in the surface description and set its **dwFlags** member to contain the desired pixel format flags—**DDPF_RGB**, **DDPF_YUV**, or both. You need not set any other pixel format values.

If you include the **DDSD_HEIGHT** and **DDSD_WIDTH** flags in the **DDSURFACEDESC2** structure, you can specify the desired dimensions in the **dwHeight** and **dwWidth** members. If you exclude these flags, DirectDraw uses the dimensions of the primary surface.

The following code fragment shows what this call could look like to enumerate all valid surface characteristics for 96×96 RGB or YUV surfaces:

```
DDSURFACEDESC2 ddsd;
HRESULT         ddrval;
ZeroMemory(&ddsd, sizeof(ddsd));

ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_PIXELFORMAT |
              DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddpfPixelFormat.dwFlags = DDPF_YUV | DDPF_RGB;
ddsd.dwHeight = 96;
ddsd.dwWidth = 96;

ddrval = lpDD->EnumSurfaces(
        DDENUMSURFACES_CANBECREATED | DDENUMSURFACES_MATCH,
        &ddsd, NULL, EnumCallback);

if (ddrval != DD_OK)
    return FALSE;
```

When DirectDraw enumerates possible surfaces, it actually attempts to create a temporary surface that has the desired characteristics. If the attempt succeeds, then DirectDraw calls the provided **EnumSurfacesCallback** function with only the characteristics that worked; it does not provide the callback function with pointer to the temporary surface. Do not assume that a surface isn't supported if it isn't enumerated. DirectDraw's attempt to create a temporary surface could fail due to memory constraints that exist at the time of the call, resulting in those characteristics not being enumerated, even if the driver actually supports them.

Updating Surface Characteristics

You can update the characteristics of an existing surface by using the **IDirectDrawSurface4::SetSurfaceDesc** method. With this method, you can change the pixel format and location of a DirectDrawSurface object's surface memory to system memory that your application has explicitly allocated. This is useful as it allows a surface to use data from a previously allocated buffer without copying. The new surface memory is allocated by the client application and, as such, the client application must also deallocate it.

When calling the **IDirectDrawSurface4::SetSurfaceDesc** method, the *lpddsd* parameter must be the address of a **DDSURFACEDESC2** structure that describes the new surface memory as well as a pointer to that memory. Within the structure, you can only set the **dwFlags** member to reflect valid members for the location of the surface memory, dimensions, pitch, and pixel format. Therefore, **dwFlags** can only contain combinations of the **DDSD_WIDTH**, **DDSD_HEIGHT**, **DDSD_PITCH**, **DDSD_LPSURFACE**, and **DDSD_PIXELFORMAT** flags, which you set to indicate valid structure members.

Before you set the values in the structure, you must allocate memory to hold the surface. The size of the memory you allocate is important. Not only do you need to allocate enough memory to accommodate the surface's width and height, but you need to have enough to make room for the surface pitch, which must be a **QWORD** (8 byte) multiple. Remember, pitch is measured in bytes, not pixels.

When setting surface values in the structure, the **lpSurface** member is a pointer to the memory you allocated and the **dwHeight** and **dwWidth** members describe the surface dimensions in pixels. If you specify surface dimensions, you must fill the **IPitch** member to reflect the surface pitch as well. Pitch must be a **DWORD** multiple. Likewise, if you specify pitch, you must also specify a width value. Lastly, the **ddpfPixelFormat** member describes the pixel format for the surface. With the exception of the **lpSurface** member, if you don't specify a value for these members, the method defaults to using the value from the current surface.

There are some restrictions you must be aware of when using **IDirectDrawSurface4::SetSurfaceDesc**, some of which are common sense. For example, the **lpSurface** member of the **DDSURFACEDESC2** structure must be a valid pointer to a system memory (the method doesn't support video memory pointers at this time). Also, the **dwWidth** and **dwHeight** members must be nonzero values. Lastly, you cannot reassign the primary surface or any surfaces within the primary's flipping chain.

You can set the same memory for multiple DirectDrawSurface objects, but you must take care that the memory is not deallocated while it is assigned to any surface object.

Using the **SetSurfaceDesc** method incorrectly will cause unpredictable behavior. The DirectDrawSurface object will not deallocate surface memory that it didn't allocate. Therefore, when the surface memory is no longer needed, it is your responsibility to deallocate it. However, when **SetSurfaceDesc** is called, DirectDraw frees the original surface memory that it implicitly allocated when creating the surface.

Accessing Surface Memory Directly

You can directly access the frame buffer or off-screen surface memory by using the **IDirectDrawSurface4::Lock** method. When you call this method, the *lpDestRect* parameter is a pointer to a **RECT** structure that describes the rectangle on the surface you want to access directly. To request that the entire surface be locked, set *lpDestRect* to NULL. Also, you can specify a **RECT** that covers only a portion of the surface. Providing that no two rectangles overlap, two threads or processes can simultaneously lock multiple rectangles in a surface.

The **Lock** method fills a **DDSURFACEDESC2** structure with all the information you need to properly access the surface memory. The structure includes information about the pitch (or stride) and the pixel format of the surface, if different from the pixel format of the primary surface. When you finish accessing the surface memory, call the **IDirectDrawSurface4::Unlock** method to unlock it.

While you have a surface locked, you can directly manipulate the contents. The following list describes some tips for avoiding common problems with directly rendering surface memory:

- Never assume a constant display pitch. Always examine the pitch information returned by the **IDirectDrawSurface4::Lock** method. This pitch can vary for a number of reasons, including the location of the surface memory, the type of display card, or even the version of the DirectDraw driver. For more information, see *Width vs. Pitch*.
- Make certain you blit to unlocked surfaces. DirectDraw blit methods will fail, returning **DDERR_SURFACEBUSY** or **DDERR_LOCKEDSURFACES**, if called on a locked surface. Similarly, GDI blit functions fail without returning error values if called on a locked surface that exists in display memory.
- Limit your application's activity while a surface is locked. While a surface is locked, DirectDraw often holds the Win16Mutex (also known as the Win16Lock) so that gaining access to surface memory can occur safely. The Win16Mutex serializes access to GDI and USER dynamic-link libraries, shutting down Windows for the duration between the **IDirectDrawSurface4::Lock**

and **IDirectDrawSurface4::Unlock** calls. The **IDirectDrawSurface4::GetDC** method implicitly calls **Lock**, and the **IDirectDrawSurface4::ReleaseDC** implicitly calls **Unlock**.

- Always copy data aligned to display memory. (Windows 95 and Windows 98 use a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to DirectDraw. Copying data unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.)

Unless you include the **DDLOCK_NOSYSLOCK** flag when you call the **Lock** method, locking the surface typically causes DirectDraw to take the **Win16Mutex**. During the **Win16Mutex** all other applications, including Windows, cease execution. Since the **Win16Mutex** stops applications from executing, standard debuggers cannot be used while the lock is held. Kernel debuggers can be used during this period. DirectDraw always takes the **Win16Mutex** when locking the primary surface.

If a blit is in progress when you call **IDirectDrawSurface4::Lock**, the method will return immediately with an error, as a lock cannot be obtained. To prevent the error, use the **DDLOCK_WAIT** flag to cause the method to wait until a lock can be successfully obtained.

Locking portions of the primary surface can interfere with the display of a software cursor. If the cursor intersects the locked rectangle, it is hidden for the duration of the lock. If it doesn't intersect the rectangle, it is frozen for the duration of the lock. Neither of these effects occurs if the entire surface is locked.

Gamma and Color Controls

This section contains information about the gamma and color control interfaces used with DirectDrawSurface objects. Information is organized into the following topics:

- What Are Gamma and Color Controls?
- Using Gamma Controls
- Using Color Controls

Note

You should not attempt to use both the **IDirectDrawGammaControl** and **IDirectDrawColorControl** interfaces on a single surface. Their effects are undefined when used together.

What Are Gamma and Color Controls?

Through the gamma and color control interfaces, DirectDrawSurface objects enable you to change how the system displays the contents of the surface, without affecting the contents of the surface itself. You can think of these controls as very simple filters that DirectDraw applies to the data as it leaves a surface before being rendered on the screen. Surface objects implement the **IDirectDrawGammaControl** and **IDirectDrawColorControl** interfaces which expose methods to adjust how the surface's contents are filtered. You can retrieve a pointer to either interface by using the **IUnknown::QueryInterface** method of the target surface, specifying the *IID_IDirectDrawGammaControl* or *IID_IDirectDrawColorControl* reference identifiers.

Gamma controls, represented by the **IDirectDrawGammaControl** interface, make it possible for you to dynamically change how a surface's individual red, green, and blue levels map to the actual levels that the system displays. By setting gamma levels, you can cause the user's screen to flash colors—red when the user's character is shot, green when they pick up a new item, and so on—without blitting new images to the frame buffer to achieve the effect. Or, you might adjust color levels to apply a color bias to the images in the frame buffer. Although this interface is similar to the color control interface, this one is the easiest to use, making it the best choice for game applications. For details, see Using Gamma Controls.

The **IDirectDrawColorControl** interface allows you to control color in a surface much like the color controls you might find on a television. The similarity between **IDirectDrawColorControl** and the actual controls on a TV is no mistake—this interface is most appropriate for adjusting how broadcast video looks in an overlay surface, so it makes sense that it should provide similar control over colors. You can use color controls to allow a user to change video characteristics such as hue, saturation, contrast, and several others. For more information, see Using Color Controls.

Using Gamma Controls

The **IDirectDrawGammaControl** interface, which you retrieve by querying the surface with the *IID_IDirectDrawGammaControl* reference identifier, allows you to manipulate ramp levels that affect the red, green, and blue color components of pixels from the surface before they are sent to the digital-to-analog converter (DAC) for display. Although all surface types support the

IDirectDrawGammaControl interface, you are only allowed to adjust gamma on the primary surface. Attempts to call **IDirectDrawGammaControl::GetGammaRamp** or **IDirectDrawGammaControl::SetGammaRamp** on a surface other than the primary surface will fail.

In the following topics, this section describes the general concept of ramp levels, and provides information about working with those levels through the methods of **IDirectDrawGammaControl**:

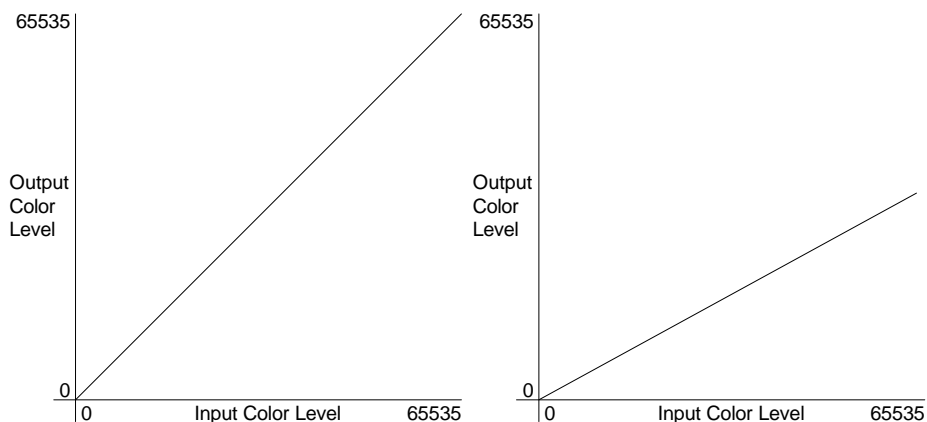
- About Gamma Ramp Levels
- Detecting Gamma Ramp Support
- Setting and Retrieving Gamma Ramp Levels

About Gamma Ramp Levels

A *gamma ramp* in DirectDraw is a term used to describe a set of values that map the level of a particular color component (red, green, blue) for all pixels in the frame buffer to new levels that are received by the digital-to-analog converter (DAC) for display on the monitor. The remapping is performed by way of three simple look-up tables, one for each color component.

Here's how it works: DirectDraw takes a pixel from the frame buffer, and looks at it in terms of its individual red, green, and blue color components. Each component is represented by a value from 0 to 65535. DirectDraw takes the original value, and uses it to index into an 256-element array (the ramp), where each element contains a value that replaces the original one. DirectDraw performs this "look-up and replace" process for each color component of each pixel within the frame buffer, thereby changing the final colors for all of the on-screen pixels.

It's handy to visualize the ramp values by graphing them. The left graph of the two following graphs shows a ramp that doesn't modify colors at all, and the right graph shows a ramp that imposes a negative bias to the color component to which it is applied.



The array elements for the graph on the left would contain values identical to their index (0 in the element at index 0, and 65535 at index 255). This type of ramp is the default, as it doesn't change the input values before they're displayed. The right graph is a little more interesting; its ramp contains values that range from 0 in the first element to 32768 in the last element, with values ranging relatively uniformly in between. The effect is that the color component that uses this ramp appears muted on the display. You are not limited to using linear graphs; if your application needs to assign arbitrary mapping, it's free to do so. You can even set the entries to all zeroes to leech a particular color component completely from the display.

Detecting Gamma Ramp Support

You can determine whether the hardware supports dynamic gamma ramp adjustment by calling the **IDirectDraw4::GetCaps** method. After the call, if the `DDCAPS2_PRIMARYGAMMA` flag is present in the `dwFlags2` member of the associate **DDCAPS** structure, the hardware supports dynamic gamma ramps. DirectDraw does not attempt to emulate this feature, so if the hardware doesn't support it, you can't use it.

Setting and Retrieving Gamma Ramp Levels

Gamma ramp levels are effectively look-up tables that DirectDraw uses to map the frame buffer color components to new levels that will be displayed. For more information, see About Gamma Ramp Levels. You set and retrieve ramp levels for the primary surface by calling the **IDirectDrawGammaControl::SetGammaRamp** and **IDirectDrawGammaControl::GetGammaRamp** methods. Both methods accept two parameters, but the first parameter is reserved for future use, and should be set to zero. The second parameter, *lpRampData*, is the address of a **DDGAMMARAMP** structure. The **DDGAMMARAMP** structure contains three 256-element arrays of **WORDS**, one array each to contain the red, green, and blue gamma ramps.

You can include the `DDSGR_CALIBRATE` value when calling the **IDirectDrawGammaControl::SetGammaRamp** to invoke the calibrator when setting new gamma levels. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently. Setting a calibrated gamma ramp will provide a consistent and absolute gamma value for the viewer, regardless of the display adapter and monitor.

Not all systems support gamma calibration. To determine if gamma calibration is supported, call **IDirectDraw4::GetCaps**, and examine the **dwCaps2** member of the associated **DDCAPS** structure after the method returns. If the `DDCAPS2_CANCALIBRATEGAMMA` capability flag is present, then gamma calibration is supported.

When setting new ramp levels, keep in mind that the levels you set in the arrays are only used when your application is in full-screen, exclusive mode. Whenever your application changes to normal mode, the ramp levels are set aside, taking effect again when the application reinstates full-screen mode. In addition, remember that you cannot set ramp levels for any surface other than the primary.

Note

Those very familiar with the Win32® API might wonder why DirectDraw exposes an interface like **IDirectDrawGammaControl**, when Win32 offers the **GetDeviceGammaRamp** and **SetDeviceGammaRamp** functions for the same surfaces. Although the Win32 API includes these functions, they do not always succeed on all Windows platforms like the methods of the **IDirectDrawGammaControl** interface.

Using Color Controls

You set and retrieve surface color controls through the **IDirectDrawColorControl** interface, which can be retrieved by querying the `DirectDrawSurface` object using the `IID_IDirectDrawColorControl` reference identifier.

Color control information is represented by a **DDCOLORCONTROL** structure, which is used with both methods of the interface, **IDirectDrawColorControl::SetColorControls** and **IDirectDrawColorControl::GetColorControls**. The first structure member, **dwSize**, should be set to the size of the structure, in bytes, before you use it. How you use the next member, **dwFlags**, depends on whether you are setting or retrieving color controls. If you are setting new color controls, set **dwFlags** to a combination of the appropriate flags to indicate which of the other structure members contain valid data that you've set. However, when retrieving color controls, you don't need to set the **dwFlags** before using it—it will contain flags telling you which members are valid after the **IDirectDrawColorControl::GetColorControls** method returns.

The remaining **DDCOLORCONTROL** structure members can contain values that describe the brightness, contrast, hue, saturation, sharpness, gamma, and whether color is used. Note that the structure contains information about gamma correction. This is a single gamma value that affects overall brightness, and it should not be confused with the gamma adjustment features provided through the **IDirectDrawGammaControl** interface.

Overlay Surfaces

This section contains information about DirectDraw overlay surface support. The following topics are discussed:

- Overlay Surface Overview
- Significant DDCAPS Members and Flags
- Source and Destination Rectangles
- Boundary and Size Alignment
- Minimum and Maximum Stretch Factors
- Overlay Color Keys
- Positioning Overlay Surfaces
- Creating Overlay Surfaces
- Overlay Z-Orders
- Flipping Overlay Surfaces

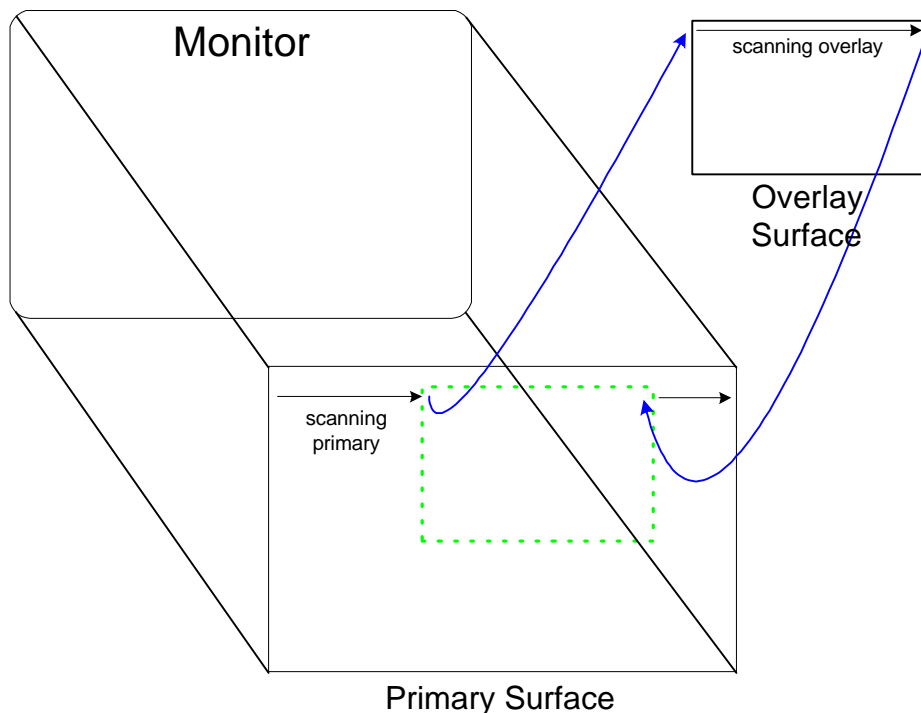
For information about implementing overlay surfaces, see Tutorial 6: Using Overlay Surfaces.

Overlay Surface Overview

Overlay surfaces, casually referred to as overlays, are surfaces with special hardware-supported capabilities. Overlay surfaces are frequently used to display live video, recorded video, or still bitmaps over the primary surface without blitting to the primary

surface or changing the primary surface's contents in any way. Overlay surface support is provided entirely by the hardware; DirectDraw supports any capabilities as reported by the display device driver. DirectDraw does not emulate overlay surfaces.

An overlay surface is analogous to a clear piece of plastic that you draw on and place in front of the monitor. When the overlay is in front of the monitor, you can see both the overlay and the contents of the primary surface together, but when you remove it, the primary surface's contents are unchanged. In fact, the mechanics of overlays work much like the clear plastic analogy. When you display an overlay surface, you're telling the device driver where and how you want it to be visible. While the display device paints scan lines to the monitor, it checks the location of each pixel in the primary surface to see if an overlay should be visible there instead. If so, the display device substitutes data from the overlay surface for the corresponding pixel, as shown in the following illustration.



By using this method, the display adapter produces a composite of the primary surface and the overlay on the monitor, providing transparency and stretching effects, without modifying the contents of either surface. The composited surfaces are injected into the video stream and sent directly to the monitor. Because this on-the-fly processing and pixel substitution is handled at the hardware level, no noticeable performance loss occurs when displaying overlays. Additionally, this method makes it possible to seamlessly composite primary and overlay surfaces with different pixel formats.

You create overlay surfaces by calling the **IDirectDraw4::CreateSurface** method, specifying the **DDSCAPS_OVERLAY** flag in the associated **DDSCAPS2** structure. Overlay surfaces can only be created in video memory, so you must also include the **DDSCAPS_VIDEOMEMORY** flag. As with other types of surfaces, by including the appropriate flags you can create either a single overlay or a flipping chain made up of multiple overlay surfaces.

Significant DDCAPS Members and Flags

You can retrieve information about the supported overlay features by calling the **IDirectDraw4::GetCaps** method. The method fills a **DDCAPS** structure with information describing all features.

When reporting hardware features, the device driver sets flags in the **dwCaps** structure member to indicate when a given type of restriction is enforced by the hardware. After retrieving the driver capabilities, examine the flags in the **dwCaps** member for information about which restrictions apply. The **DDCAPS** structure contains nine members that carry information describing hardware restrictions for overlay surfaces. The following table lists the overlay related members and their corresponding flags:

Member	Flag
dwMaxVisibleOverlays	This member is always valid
dwCurrVisibleOverlays	This member is always valid
dwAlignBoundarySrc	DDCAPS_ALIGNBOUNDARYSRC
dwAlignSizeSrc	DDCAPS_ALIGNSIZESRC

dwAlignBoundaryDest	DDCAPS_ALIGNBOUNDARYDEST
dwAlignSizeDest	DDCAPS_ALIGNSIZEDEST
dwAlignStrideAlign	DDCAPS_ALIGNSTRIDE
dwMinOverlayStretch	DDCAPS_OVERLAYSTRETCH
dwMaxOverlayStretch	DDCAPS_OVERLAYSTRETCH

The **dwMaxVisibleOverlays** and **dwCurrVisibleOverlays** members carry information about the maximum number of overlays the hardware can display, and how many of them are currently visible.

Additionally, the hardware reports rectangle position and size alignment restrictions in the **dwAlignBoundarySrc**, **dwAlignSizeSrc**, **dwAlignBoundaryDest**, **dwAlignSizeDest**, and **dwAlignStrideAlign** members. The values in these members dictate how you must size and position source and destination rectangles when displaying overlay surfaces. For more information, see Source and Destination Rectangles and Boundary and Size Alignment.

Also, the hardware reports information about stretch factors in the **dwMinOverlayStretch** and **dwMaxOverlayStretch** members. For more information, see Minimum and Maximum Stretch Factors.

Source and Destination Rectangles

To display an overlay surface, you call the overlay surface's **IDirectDrawSurface4::UpdateOverlay** method, specifying the **DDOVER_SHOW** flag in the *dwFlags* parameter. The method requires you to specify a source and destination rectangle in the *lpSrcRect* and *lpDestRect* parameters. The source rectangle describes a rectangle on the overlay surface that will be visible on the primary surface. To request that the method use the entire surface, set the *lpSrcRect* parameter to **NULL**. The destination rectangle describes a portion of the primary surface on which the overlay surface will be displayed.

Source and destination rectangles do not need to be the same size. You can often specify a destination rectangle smaller or larger than the source rectangle, and the hardware will shrink or stretch the overlay appropriately when it is displayed.

To successfully display an overlay surface, you might need to adjust the size and position of both rectangles. Whether this is necessary depends on the restrictions imposed by the device driver. For more information, see Boundary and Size Alignment and Minimum and Maximum Stretch Factors.

Boundary and Size Alignment

Due to various hardware limitations, some device drivers impose restrictions on the position and size of the source and destination rectangles used to display overlay surfaces. To find out which restrictions apply for a device, call the **IDirectDraw4::GetCaps** method and then examine the overlay-related flags in the **dwCaps** member of the **DDCAPS** structure. The following table shows the members and flags specific to boundary and size alignment restrictions:

Category	Flag	Member
Boundary (position) restrictions	DDCAPS_ALIGNBOUNDARYSRC	dwAlignBoundarySrc
	DDCAPS_ALIGNBOUNDARYDEST	dwAlignBoundaryDest
Size restrictions	DDCAPS_ALIGNSIZESRC	dwAlignSizeSrc
	DDCAPS_ALIGNSIZEDEST	dwAlignSizeDest

There are two types of restrictions, boundary restrictions and size restrictions. Both types of restrictions are expressed in terms of pixels (not bytes) and can apply to the source and destination rectangles. Also, these restrictions can vary depending on the pixel formats of the overlay and primary surface.

Boundary restrictions affect where you can position a source or destination rectangle. The values in the **dwAlignBoundarySrc** and **dwAlignBoundaryDest** members tell you how to align the top left corner of the corresponding rectangle. The x-coordinate of the top left corner of the rectangle (the **left** member of the **RECT** structure), must be a multiple of the reported value.

Size restrictions affect the valid widths for source and destination rectangles. The values in the **dwAlignSizeSrc** and **dwAlignSizeDest** members tell you how to align the width, in pixels, of the corresponding rectangle. Your rectangles must have a pixel width that is a multiple of the reported value. If you stretch the rectangle to comply with a minimum required stretch factor, be sure that the stretched rectangle is still size aligned. After stretching the rectangle, align its width by rounding up, not down, so you preserve the minimum stretch factor. For more information, see Minimum and Maximum Stretch Factors.

Minimum and Maximum Stretch Factors

Due to hardware limitations, some devices restrict how wide a destination rectangle can be compared with the corresponding source rectangle. DirectDraw communicates these restrictions as stretch factors. A stretch factor is the ratio between the widths of the source and destination rectangles. If the driver provides information about stretch factors, it sets the `DDCAPS_OVERLAYSTRETCH` flag in the `DDCAPS` structure after you call the `IDirectDraw4::GetCaps` method. Note that stretch factors are reported multiplied by 1000, so a value of 1300 actually means 1.3 (and 750 would be 0.75).

Devices that do not impose limits on stretching or shrinking an overlay destination rectangle often report a minimum and maximum stretch factor of 0.

The minimum stretch factor tells you how much wider or narrower than the source rectangle the destination rectangle needs to be. If the minimum stretch factor is greater than 1000, then you must increase the destination rectangle's width by that ratio. For instance, if the driver reports 1300, you must make sure that the destination rectangle's width is at least 1.3 times the width of the source rectangle. Similarly, a minimum stretch factor less than 1000 indicates that the destination rectangle can be smaller than the source rectangle by that ratio.

The maximum stretch factor tells the maximum amount you can stretch the width of the destination rectangle. For example, if the maximum stretch factor is 2000, you can specify destination rectangles that are up to, but not wider than, twice the width of the source rectangle. If the maximum stretch factor is less than 1000, then you must shrink the width of the destination rectangle by that ratio to be able to display the overlay.

After stretching, the destination rectangle must conform to any size alignment restrictions the device might require. Therefore, it's a good idea to stretch the destination rectangle before adjusting it to be size aligned. For more information, see [Boundary and Size Alignment](#).

Hardware does not require that you adjust the height of destination rectangles. You can increase a destination rectangle's height to preserve aspect ratio without negative effects.

Overlay Color Keys

Like other types of surfaces, overlay surfaces use source and destination color keys for controlling transparent blit operations between surfaces. Because overlay surfaces are not displayed by blitting, there needs to be a different way to control how an overlay surface is displayed over the primary surface when you call the `IDirectDrawSurface4::UpdateOverlay` method. This need is filled by overlay color keys. Overlay color keys, like their blit-related counterparts, have a source version and a destination version that you set by calling the `IDirectDrawSurface4::SetColorKey` method. (For more information, see [Setting Color Keys](#).) You use the `DDCKEY_SRCOVERLAY` or `DDCKEY_DESTOVERLAY` flags to set a source or destination overlay color key. Overlay surfaces can employ blit and overlay color keys together to control blit operations and overlay display operations appropriately; the two types of color keys do not conflict with one another.

The `IDirectDrawSurface4::UpdateOverlay` method uses the source overlay color key to determine which pixels in the overlay surface should be considered transparent, allowing the contents of the primary surface to show through. Likewise, the method uses the destination overlay color key to determine the parts of the primary surface that will be covered up by the overlay surface when it is displayed. The resulting visual effect is the same as that created by blit-related color keys.

Positioning Overlay Surfaces

After initially displaying an overlay by calling the `IDirectDrawSurface4::UpdateOverlay` method, you can update the destination rectangle's by calling the `IDirectDrawSurface4::SetOverlayPosition` method.

Make sure that the positions you specify comply with any boundary alignment restrictions enforced by the hardware. For more information, see [Boundary and Size Alignment](#). Also remember that `SetOverlayPosition` doesn't perform clipping for you; using coordinates that would potentially make the overlay run off the edge of the target surface will cause the method to fail, returning `DDERR_INVALIDPOSITION`.

Creating Overlay Surfaces

Like all surfaces, you create an overlay surface by calling the `IDirectDraw4::CreateSurface` method. To create an overlay, include the `DDSCAPS_OVERLAY` flag in the associated `DDSCAPS2` structure.

Overlay support varies widely across display devices. As a result, you cannot be sure that a given pixel format will be supported by most drivers and must therefore be prepared to work with a variety of pixel formats. You can request information about the non-RGB formats that a driver supports by calling the `IDirectDraw4::GetFourCCCodes` method.

When you attempt to create an overlay surface, it is advantageous to try creating a surface with the most desirable pixel format, falling back on other pixel formats if a given pixel format isn't supported.

You can create overlay surface flipping chains. For more information, see [Creating Complex Surfaces and Flipping Chains](#).

Overlay Z-Orders

Overlay surfaces are assumed to be on top of all other screen components, but when you display multiple overlay surfaces, you need some way to visually organize them. DirectDraw supports *overlay z-ordering* to manage the order in which overlays clip each other. Z-order values represent conceptual distances from the primary surface toward the viewer. They range from 0, which is just on top of the primary surface, to 4 billion, which is as close to the viewer as possible, and no two overlays can share the same z-order. You set z-order values by calling the **IDirectDrawSurface4::UpdateOverlayZOrder** method.

Destination color keys are affected only by the bits on the primary surface, not by overlays occluded by other overlays. Source color keys work on an overlay whether or not a z-order was specified for the overlay.

Overlays without a specified z-order are assumed to have a z-order of 0. Overlays that do not have a specified z-order behave in unpredictable ways when overlaying the same area on the primary surface.

A DirectDraw object does not track the z-orders of overlays displayed by other applications.

Note

You can ensure proper clipping of multiple overlay surfaces by calling **UpdateOverlayZOrder** in response to WM_KILLFOCUS messages. When you receive this message, set your overlay surface to the rearmost z-order position by calling the **UpdateOverlayZOrder** method with the *dwFlags* parameter set to DDOVERZ_SENDBACK.

Flipping Overlay Surfaces

Like other types of surfaces, you can create overlay flipping chains. After creating a flipping chain of overlays, call the **IDirectDrawSurface4::Flip** method to flip between them. For more information, see [Flipping Surfaces](#).

Software decoders displaying video with overlay surfaces can use the DDFLIP_ODD and DDFLIP_EVEN flags when calling the **Flip** method to use features that reduce motion artifacts. If the driver supports odd-even flipping, the DDSCAPS2_CANFLIPODDEVEN flag will be set in the **DDSCAPS** structure after retrieving driver capabilities. If DDSCAPS2_CANFLIPODDEVEN is set, you can include the DDOVER_BOB flag when calling the **IDirectDrawSurface4::UpdateOverlay** method to inform the driver that you want it to use the "Bob" algorithm to minimize motion artifacts. Later, when you call **Flip** with the DDFLIP_ODD or DDFLIP_EVEN flag, the driver will automatically adjust the overlay source rectangle to compensate for jittering artifacts.

If the driver doesn't set the DDSCAPS2_CANFLIPODDEVEN flag when you retrieve hardware capabilities, **UpdateOverlay** will fail if you specify the DDOVER_BOB flag.

For more information about the Bob algorithm, see [Solutions to Common Video Artifacts](#).

Compressed Texture Surfaces

A surface can contain a bitmap to be used for texturing 3-D objects. When creating the surface you must specify the DDSCAPS_TEXTURE flag in the **dwFlags** member of the **DDSCAPS** structure.

For more information on the use of textures in Direct3D Immediate Mode, see [Textures](#).

In order to reduce the amount of memory consumed by textures, DirectDraw supports the compression of texture surfaces.

Some Direct3D devices support compressed texture surfaces natively. On such devices, once you have created a compressed surface and loaded the data into it, the surface can be used in Direct3D just like any other texture surface. Direct3D handles decompression when the texture is mapped to a 3-D object.

Other devices do not support compressed texture surfaces natively. When using such devices, you may still find it useful to use compressed surfaces to represent textures on disk or for textures that are loaded into memory but not currently being used. You can use DirectDraw to convert the compressed textures to an uncompressed format before giving the texture to Direct3D.

For more information on texture compression in DirectDraw, see the following topics:

- [Creating Compressed Textures](#)
- [Decompressing Compressed Textures](#)

- Transparency in Blits to Compressed Textures
- Compressed Texture Formats

For information on using compressed textures in Direct3D Immediate Mode, see Texture Compression.

Creating Compressed Textures

To describe a compressed texture surface in the **DDSURFACEDESC2** structure when creating the surface, you must include the following steps:

- Specify the **DDSCAPS_TEXTURE** flag in the **dwFlags** member of the **DDSCAPS** structure, just as you would for any texture.
- Set the **dwFourCC** member of the **DDPIXELFORMAT** structure to one of the DXT codes described later.
- Include **DDPF_FOURCC** in the **dwFlags** member of **DDPIXELFORMAT**. Do not set the **DDPF_RGB** flag.
- Specify a width and height that are a multiple of 4 pixels.

There are two ways to load image data into a compressed texture surface:

- Create a regular RGB or ARGB surface and load a normal bitmap into it, then use **IDirectDrawSurface4::Blt** or **IDirectDrawSurface4::BltFast** to blit from the uncompressed surface to the compressed surface. DirectDraw does the compression for you.
- Load the compressed data from a file and copy it directly into the surface memory. (See Accessing Surface Memory Directly.) You can create and convert compressed texture (DDS) files using the DirectX Texture Tool (Dxtex.exe) supplied with the Programmer's Reference. You can also create your own DDS files and either copy the data from compressed surfaces or else use your own routines to convert regular bitmap data to one of the compressed formats.

Note

When you call **IDirectDrawSurface4::Lock** or **IDirectDrawSurface4::GetSurfaceDesc** on a compressed surface, the **DDSD_LINEARSIZE** flag is set in the **dwFlags** member of the **DDSURFACEDESC** structure, and the **dwLinearSize** member contains the number of bytes allocated to contain the compressed surface data. The **dwLinearSize** parameter resides in a union with the **IPitch** parameter, so these parameters are mutually exclusive, as are the flags **DDSD_LINEARSIZE** and **DDSD_PITCH**.

The advantage of this behavior is that an application can copy the contents of a compressed surface to a file without having to calculate for itself how much storage is required for a surface of a particular width and height in the specific format.

The following table shows the five types of compressed textures. For more information on how the data is stored (you need to know this only if you are writing your own compression routines) see Compressed Texture Formats.

FOURCC	Description	Alpha premultiplied?
DXT1	Opaque / one-bit alpha	n/a
DXT2	Explicit alpha	Yes
DXT3	Explicit alpha	No
DXT4	Interpolated alpha	Yes
DXT5	Interpolated alpha	No

Note

When you blit from a non-premultiplied format to a premultiplied format, DirectDraw scales the colors based on the alpha values. Blitting from a premultiplied format to a non-premultiplied format is not supported. If you try to blit from a premultiplied-alpha source to a non-premultiplied-alpha destination, the method will return **DDERR_INVALIDPARAMS**. If you blit from a premultiplied-alpha source to a destination that has no alpha, the source color components, which have been scaled by alpha, will be copied as is.

Decompressing Compressed Textures

As with compressing a texture surface, decompressing a compressed texture is performed through DirectDraw blitting services. The HEL performs decompressing blits between system memory surfaces, so these always supported. Likewise, the HEL always

performs blits for compressed managed textures (the DDSCAPS2_TEXTUREMANAGE capability). For other situations, the restrictions discussed in the following paragraphs apply.

If the driver supports the creation of compressed video-memory surfaces, then the driver can also perform decompressing blits from a compressed video-memory surface to an uncompressed video or system memory surface, so long as the destination surface has the DDSCAPS_OFFSCREENPLAIN capability.

Blits from compressed system-memory surfaces to uncompressed video-memory surfaces are largely unsupported and should not be attempted, even when the driver supports compressed textures. This does not mean that it is impossible to decompress a compressed system-memory surface and move its contents into a video memory surface; it merely requires an additional step:

0 To decompress a system memory surface into video memory:

1. Create an uncompressed, offscreen-plain, surface in system memory of the desired dimensions and pixel format.
2. Blit from the compressed system-memory surface to the uncompressed system-memory surface. (The DirectDraw HEL performs decompression in this case.)
3. Blit the uncompressed surface to the uncompressed video-memory surface.

Transparency in Blits to Compressed Textures

DirectDraw provides a special trick for creating compressed textures with alpha from plain RGB surfaces. If a source color key is provided on the source RGB surface, DirectDraw assigns an alpha value of 0 to all pixels of that color in the destination. This technique is especially useful for creating DXT1 textures, since they effectively have only 1 bit of alpha information per pixel.

Note

There are no flags that control this behavior. If you do not want any transparency in your compressed texture, do not set a source color key on the source surface.

Compressed Texture Formats

This section contains information on the internal organization of compressed texture formats. You don't need these details in order to use compressed textures, because DirectDraw handles conversion to and from compressed formats. However, you might find this information useful if you want to operate on compressed surface data directly.

DirectDraw uses a compression format that divides texture maps into 4x4 texel blocks. If the texture contains no transparency (is opaque), or if the transparency is specified by a one-bit alpha, an 8-byte block represents the texture map block. If the texture map does contain transparent texels, using an alpha channel, a 16-byte block represents it.

These two types of format are discussed in the following sections:

- Opaque and One-bit Alpha Textures
- Textures with Alpha Channels

Note

Any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels.

When 128-bit blocks are used, then the alpha channel must be specified in either explicit (format DXT2 or DXT3) or interpolated mode (format DXT4 or DXT5) for the entire texture. Note that as with color, once interpolated mode is selected then either 8 interpolated alphas or 6 interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.

Opaque and One-bit Alpha Textures

Texture format DXT1 is for textures that are opaque or have a single transparent color.

For each opaque or one-bit alpha block, two 16-bit values (RGB 5:6:5 format) and a 4x4 bitmap with 2-bits-per-pixel are stored. This totals 64 bits for 16 texels, or 4-bits-per-texel. In the block bitmap, there are two bits per texel to select between the four colors, two of which are stored in the encoded data. The other two colors are derived from these stored colors by linear interpolation.

The one-bit alpha format is distinguished from the opaque format by comparing the two 16-bit color values stored in the block. They are treated as unsigned integers. If the first color is greater than the second, it implies that only opaque texels are defined. This means four colors will be used to represent the texels. In four-color encoding, there are two derived colors and all four colors are equally distributed in RGB color space. This format is analogous to RGB 5:6:5 format. Otherwise, for one-bit alpha transparency, three colors are used and the fourth is reserved to represent transparent texels.

In three-color encoding, there is one derived color and the fourth two-bit code is reserved to indicate a transparent texel (alpha information). This format is analogous to RGBA 5:5:5:1, where the final bit is used for encoding the alpha mask.

The following piece of pseudo-code illustrates the algorithm for deciding whether three- or four-color encoding is selected:

```

if (color_0 > color_1)
{
    // Four-color block: derive the other two colors.
    // 00 = color_0, 01 = color_1, 10 = color_2, 11 = color_3
    // These two bit codes correspond to the 2-bit fields
    // stored in the 64-bit block.
    color_2 = (2 * color_0 + color_1) / 3;
    color_3 = (color_0 + 2 * color_1) / 3;
}
else
{
    // Three-color block: derive the other color.
    // 00 = color_0, 01 = color_1, 10 = color_2,
    // 11 = transparent.
    // These two bit codes correspond to the 2-bit fields
    // stored in the 64-bit block.
    color_2 = (color_0 + color_1) / 2;
    color_3 = transparent;
}

```

The following tables show the memory layout for the 8-byte block. It is assumed that the first index corresponds to the y-coordinate and the second corresponds to the x-coordinate. For example, Texel[1][2] refers to the texture map pixel at (x,y) = (2,1).

Here is the memory layout for the 8-byte (64-bit) block:

Word address	16-bit word
0	Color_0
1	Color_1
2	Bitmap Word_0
3	Bitmap Word_1

Color_0 and Color_1 (colors at the two extremes) are laid out as follows:

Bits	Color
4:0 (LSB)	Blue color component
10:5	Green color component
15:11	Red color component

Bitmap Word_0 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[0][0]
3:2	Texel[0][1]
5:4	Texel[0][2]
7:6	Texel[0][3]
9:8	Texel[1][0]
11:10	Texel[1][1]

13:12	Texel[1][2]
15:14 (MSB)	Texel[1][3]

Bitmap Word_1 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[2][0]
3:2	Texel[2][1]
5:4	Texel[2][2]
7:6	Texel[2][3]
9:8	Texel[3][0]
11:10	Texel[3][1]
13:12	Texel[3][2]
15:14 (MSB)	Texel[3][3]

Example of Opaque Color Encoding

As an example of opaque encoding, we will assume that the colors red and black are at the extremes. We will call red color_0 and black color_1. There will be four interpolated colors that form the uniformly distributed gradient between them. To determine the values for the 4x4 bitmap, the following calculations are used:

```
00 ? color_0
01 ? color_1
10 ? 2/3 color_0 + 1/3 color_1
11 ? 1/3 color_0 + 2/3 color_1
```

Example of One-bit Alpha Encoding

This format is selected when the unsigned 16-bit integer, color_0, is less than the unsigned 16-bit integer, color_1. An example of where this format could be used is leaves on a tree to be shown against a blue sky. Some texels could be marked as transparent while three shades of green are still available for the leaves. Two of these colors fix the extremes, and the third color is an interpolated color.

The bitmap encoding for the colors and the transparency is determined using the following calculations:

```
00 ? color_0
01 ? color_1
10 ? 1/2 color_0 + 1/2 color_1
11 ? Transparent
```

Textures with Alpha Channels

There are two ways to encode texture maps that exhibit more complex transparency. In each case, a block that describes the transparency precedes the 64-bit block already described. The transparency is either represented as a 4x4 bitmap with four bits per pixel (explicit encoding), or with fewer bits and linear interpolation analogous to what is used for color encoding.

The transparency block and the color block are laid out as follows:

Word Address	64-bit Block
3:0	Transparency block
7:4	Previously described 64-bit block

Explicit Texture Encoding

For explicit texture encoding (DXT2 and DXT3 formats), the alpha components of the texels that describe transparency are encoded in a 4x4 bitmap with 4 bits per texel. These 4 bits can be achieved through a variety of means such as dithering or by simply using the 4 most significant bits of the alpha data. However they are produced, they are used just as they are, without any form of interpolation.

Note

DirectDraw's compression method uses the 4 most significant bits.

The following tables illustrate how the alpha information is laid out in memory, for each 16-bit word.

This is the layout for Word 0:

Bits	Alpha
3:0 (LSB)	[0][0]
7:4	[0][1]
11:8	[0][2]
15:12 (MSB)	[0][3]

This is the layout for Word 1:

Bits	Alpha
3:0 (LSB)	[1][0]
7:4	[1][1]
11:8	[1][2]
15:12 (MSB)	[1][3]

This is the layout for Word 2:

Bits	Alpha
3:0 (LSB)	[2][0]
7:4	[2][1]
11:8	[2][2]
15:12 (MSB)	[2][3]

This is the layout for Word 3:

Bits	Alpha
3:0 (LSB)	[3][0]
7:4	[3][1]
11:8	[3][2]
15:12 (MSB)	[3][3]

Three-Bit Linear Alpha Interpolation

The encoding of transparency for the DXT4 and DXT5 formats is based on a concept similar to the linear encoding used for color. Two 8-bit alpha values and a 4x4 bitmap with three bits per pixel are stored in the first eight bytes of the block. The representative alpha values are used to interpolate intermediate alpha values. Additional information is available in the way the two alpha values are stored. If alpha_0 is greater than alpha_1, then six intermediate alpha values are created by the interpolation. Otherwise, four intermediate alpha values are interpolated between the specified alpha extremes. The two additional implicit alpha values are 0 (fully transparent) and 255 (fully opaque).

The following pseudo-code illustrates this algorithm:

```
// 8-alpha or 6-alpha block?
if (alpha_0 > alpha_1) {
    // 8-alpha block: derive the other 6 alphas.
    // 000 = alpha_0, 001 = alpha_1, others are interpolated
    alpha_2 = (6 * alpha_0 + alpha_1) / 7; // bit code 010
    alpha_3 = (5 * alpha_0 + 2 * alpha_1) / 7; // Bit code 011
    alpha_4 = (4 * alpha_0 + 3 * alpha_1) / 7; // Bit code 100
    alpha_5 = (3 * alpha_0 + 4 * alpha_1) / 7; // Bit code 101
    alpha_6 = (2 * alpha_0 + 5 * alpha_1) / 7; // Bit code 110
    alpha_7 = (alpha_0 + 6 * alpha_1) / 7; // Bit code 111
}
else { // 6-alpha block: derive the other alphas.
```

```

// 000 = alpha_0, 001 = alpha_1, others are interpolated
alpha_2 = (4 * alpha_0 + alpha_1) / 5; // Bit code 010
alpha_3 = (3 * alpha_0 + 2 * alpha_1) / 5; // Bit code 011
alpha_4 = (2 * alpha_0 + 3 * alpha_1) / 5; // Bit code 100
alpha_5 = (alpha_0 + 4 * alpha_1) / 5; // Bit code 101
alpha_6 = 0; // Bit code 110
alpha_7 = 255; // Bit code 111
}

```

The memory layout of the alpha block is as follows:

Byte	Alpha
0	Alpha_0
1	Alpha_1
2	[0][2] (2 LSBs), [0][1], [0][0]
3	[1][1] (1 LSB), [1][0], [0][3], [0][2] (1 MSB)
4	[1][3], [1][2], [1][1] (2 MSBs)
5	[2][2] (2 LSBs), [2][1], [2][0]
6	[3][1] (1 LSB), [3][0], [2][3], [2][2] (1 MSB)
7	[3][3], [3][2], [3][1] (2 MSBs)

Private Surface Data

You can store any kind of application-specific data with a surface. For example, a surface representing a map in a game might contain information about terrain.

A surface can have more than one private data buffer. Each buffer is identified by a GUID which you supply when attaching the data to the surface.

To store private surface data, you use the **IDirectDrawSurface4::SetPrivateData** method, passing in a pointer to the source buffer, the size of the data, and an application-defined GUID for the data. Optionally, the source data can exist in the form of a COM object; in this case, you pass a pointer to the object's **IUnknown** interface pointer and you set the `DDSPD_IUNKNOWNPOINTER` flag. Another flag, `DDSPD_VOLATILE`, indicates that the data being attached to the surface is valid only as long as the contents of the surface do not change. (See Surface Uniqueness Values.)

SetPrivateData allocates an internal buffer for the data and copies it. You can then safely free the source buffer or object. The internal buffer or interface reference is released when **IDirectDrawSurface4::FreePrivateData** is called. This happens automatically when the surface is freed.

To retrieve private data for a surface, you must allocate a buffer of the correct size and then call the **IDirectDrawSurface4::GetPrivateData** method, passing the GUID that was assigned to the data by **SetPrivateData**. You are responsible for freeing any dynamic memory you use for this buffer. If the data is a COM object, this method retrieves the **IUnknown** pointer.

If you don't know how big a buffer to allocate, first call **GetPrivateData** with zero in *lpcbBufferSize*. If the method fails with `DDERR_MOREDATA`, it returns the necessary number of bytes in *lpcbBufferSize*.

Surface Uniqueness Values

The uniqueness value of a surface allows you to determine whether the surface has changed. When DirectDraw creates a surface, it assigns a uniqueness value, which you can retrieve and store by using the **IDirectDrawSurface4::GetUniquenessValue** method. Then, whenever you need to determine whether the surface has changed, you call the method again and compare the new value against the old one. If it's different, the surface has changed.

The actual value returned by **GetUniquenessValue** is irrelevant, unless it is 0. DirectDraw assigns this value to a surface when it knows that the surface might be changed by some process beyond its control. When **GetUniquenessValue** returns 0, you know only that the state of the surface is indeterminate.

To force the uniqueness value for a surface to change, an application can use the **IDirectDrawSurface4::ChangeUniquenessValue** method. This method could be called, for example, by an application or

component that changed the private data for a surface without changing the surface itself, and wished to notify some other process of the change. Most applications, however, never need to change the uniqueness value.

Using Non-local Video Memory Surfaces

DirectDraw supports the Accelerated Graphics Port (AGP) architecture for creating surfaces in non-local video memory. On AGP-equipped systems, DirectDraw will use non-local video memory if local video memory is exhausted or if non-local video memory is explicitly requested, depending on the type of AGP implementation that is in place.

Currently, there are two implementations of the AGP architecture, known as the "execute model" and the "DMA model." In the execute model implementation, the display device supports the same features for non-local video memory surfaces and local video memory surfaces. As a result, when you retrieve hardware capabilities by calling the **IDirectDraw4::GetCaps** method, the blit-related flags in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure will be identical to those for local video memory. Under the execute model, if local video memory is exhausted, DirectDraw will automatically fall back on non-local video memory unless the caller specifically requests otherwise.

In the DMA model implementation, support for blitting and texturing from non-local video memory surfaces is limited. When the display device uses the DMA model, the **DDCAPS2_NONLOCALVIDMEMCAPS** flag will be set in the **dwCaps2** member when you retrieve device capabilities. In the DMA model, the blit-related flags included in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure describe the features that are supported; these features will often be a smaller subset of those supported for local video memory surfaces. Under the DMA model, when local video memory is exhausted, DirectDraw will automatically fall back on non-local video memory for texture surfaces only, unless the caller had explicitly requested local video memory. Texture surfaces are the only types of surfaces that will be treated this way; all other types of surfaces cannot be created in non-local video memory unless the caller explicitly requests it.

DMA model implementations vary in support for texturing from non-local video memory surfaces. If the driver supports texturing from non-local video memory surfaces, the **D3DDEVCAPS_TEXTURENONLOCALVIDMEM** flag will be set when you retrieve the 3-D device's capabilities by calling the **IDirect3DDevice3::GetCaps** method.

Converting Color and Format

Non-RGB surface formats are described by four-character codes (FOURCC). If an application calls the **IDirectDrawSurface4::GetPixelFormat** method to request the pixel format, and the surface is a non-RGB surface, the **DDPF_FOURCC** flag will be set and the **dwFourCC** member of the **DDPIXELFORMAT** structure will be valid. If the FOURCC code represents a YUV format, the **DDPF_YUV** flag will also be set and the **dwYUVBitCount**, **dwYBitMask**, **dwUBitMask**, **dwVBitMask**, and **dwYUVAphaBitMask** members will be valid masks that can be used to extract information from the pixels.

If an RGB format is present, the **DDPF_RGB** flag will be set and the **dwRBBitCount**, **dwRBitMask**, **dwGBitMask**, **dwBBitMask**, and **dwRGBAlphaBitMask** members will be valid masks that can be used to extract information from the pixels. The **DDPF_RGB** flag can be set in conjunction with the **DDPF_FOURCC** flag if a nonstandard RGB format is being described.

During color and format conversion, two sets of FOURCC codes are exposed to the application. One set of FOURCC codes represents the capabilities of the blitting hardware; the other represents the capabilities of the overlay hardware.

For more information, see Four Character Codes (FOURCC).

Surfaces and Device Contexts

It is often convenient to mix-and-match DirectDraw and GDI services to manipulate the contents of DirectDraw surfaces. DirectDraw offers methods to enable GDI to access DirectDraw surfaces through device contexts, and to retrieve a surface given the surface's device context. This section contains the follows topics that describe these features in detail:

- Retrieving the Device Context for a Surface
- Finding a Surface with a Device Context

Retrieving the Device Context for a Surface

If you want to modify the contents of a DirectDraw surface object by using GDI functions, you must retrieve a GDI-compatible device context handle. This could be useful if you wanted to display text in a DirectDraw surface by calling the **DrawText** Win32 function, which accepts a handle to a device context as a parameter. It is possible to retrieve a GDI-compatible device context for a surface by calling the **IDirectDrawSurface4::GetDC** method for that surface. The following example shows how this might be done:

```
// For this example the lpDDS4 variable is a valid pointer
// to an IDirectDrawSurface4 interface.
```

```
HDC      hdc;
HRESULT  hr;

hr = lpDDS4->GetDC(&hdc);
if(FAILED(hr))
    return hr;

// Call DrawText, or some other GDI
// function here.

lpDDS4->ReleaseDC(hdc);
```

Note that the code calls the **IDirectDrawSurface4::ReleaseDC** method when the surface's device context is no longer needed. This step is required, because the **IDirectDrawSurface4::GetDC** method uses an internal version of the **IDirectDrawSurface4::Lock** method to lock the surface. The surface remains locked until the **IDirectDrawSurface4::ReleaseDC** method is called.

Finding a Surface with a Device Context

You can retrieve a pointer to a surface's **IDirectDrawSurface4** interface from the device context for the surface by calling the **IDirectDraw4::GetSurfaceFromDC** method. This feature might be very useful for component applications or ActiveX® controls, that are commonly given a device context to draw into at run-time, but could benefit by exploiting the features exposed by the **IDirectDrawSurface4** interface.

A device context might identify memory that isn't associated with a DirectDraw object, or the device context might identify a surface for another DirectDraw object entirely. The latter case is most likely to occur on a system with multiple monitors. If the device context doesn't identify a surface that wasn't created by that DirectDraw object, the method fails, returning **DDERR_NOTFOUND**.

The following sample code shows what a very simple scenario might look like:

```
// For this example, the hdc variable is a valid
// handle to a video memory device context, and the
// lpDD4 variable is a valid IDirectDraw4 interface pointer.

LPDIRECTDRAWSURFACE4 lpDDS4;
HRESULT hr;

hr = lpDD4->GetSurfaceFromDC(hdc, &lpDDS4);
if(SUCCEEDED(hr)) {
    // Use the surface interface.
}
else if(DDERR_NOTFOUND == hr) {
    OutputDebugString("HDC not from this DirectDraw surface\n");
}
```

Palettes

This section contains information about DirectDrawPalette objects. The following topics are discussed:

- What Are Palettes?
- Palette Types
- Setting Palettes on Nonprimary Surfaces
- Sharing Palettes
- Palette Animation

What Are Palettes?

Palettized surfaces need palettes to be meaningfully displayed. A palettized surface, also known as a color-indexed surface, is simply a collection of numbers where each number represents a pixel. The value of the number is an index into a color table that tells DirectDraw what color to use when displaying that pixel. DirectDrawPalette objects, casually referred to as *palettes*, provide you with an easy way to manage a color table. Surfaces that use a 16-bit or greater pixel format do not use palettes.

A DirectDrawPalette object represents an indexed color table that has 2, 4, 16 or 256 entries to be used with a color indexed surface. Each entry in the palette is an RGB triplet that describes the color to be used when displaying pixels within the surface. The color table can contain 16- or 24-bit RGB triplets representing the colors to be used. For 16-color palettes, the table can also contain indexes to another 256-color palette. Palettes are supported for textures, off-screen surfaces, and overlay surfaces, none of which is required to have the same palette as the primary surface.

You can create a palette by calling the **IDirectDraw4::CreatePalette** method. This method retrieves a pointer to the palette object's **IDirectDrawPalette** interface. You can use the methods of this interface to manipulate palette entries, retrieve information about the object's capabilities, or initialize the object (if you used the **CoCreateInstance** COM function to create it).

You apply a palette to a surface by calling the surface's **IDirectDrawSurface4::SetPalette** method. A single palette can be applied to multiple surfaces.

DirectDrawPalette objects reserve entry 0 and entry 255 for 8-bit palettes, unless you specify the DDPCAPS_ALLOW256 flag to request that these entries be made available to you.

You can retrieve palette entries by using the **IDirectDrawPalette::GetEntries** method, and you can change entries by using the **IDirectDrawPalette::SetEntries** method.

The Ddutil.cpp source file included with the SDK contains some handy application-defined functions for working with palettes. For more information, see the DDLoadPalette functions in that source file.

Palette Types

DirectDraw supports 1-bit (2 entry), 2-bit (4 entry), 4-bit (16 entry), and 8-bit (256 entry) palettes. A palette can only be attached to a surface that has a matching pixel format. For example, a 2-entry palette created with the DDPCAPS_1BIT flag can be attached only to a 1-bit surface created with the DDPF_PALETTEINDEXED1 flag.

Additionally, you can create palettes that don't contain a color table at all, known as *index palettes*. Instead of a color table, an index palette contains index values that represent locations in another palette's color table.

To create an indexed palette, specify the DDPCAPS_8BITENTRIES flag when calling the **IDirectDraw4::CreatePalette** method. For example, to create a 4-bit indexed palette, specify both the DDPCAPS_4BIT and DDPCAPS_8BITENTRIES flags. When you create an indexed palette, you pass a pointer to an array of bytes rather than a pointer to an array of **PALETTEENTRY** structures. You must cast the pointer to the array of bytes to an **LPPALETTEENTRY** type when you use the **IDirectDraw4::CreatePalette** method.

Note that DirectDraw does not dereference index palette entries during blit operations.

Setting Palettes on Nonprimary Surfaces

Palettes can be attached to any palettized surface (primary, back buffer, off-screen plain, or texture map). Only those palettes attached to primary surfaces will have any effect on the system palette. It is important to note that DirectDraw blits never perform color conversion; any palettes attached to the source or destination surface of a blit are ignored.

Nonprimary surface palettes are intended for use by Direct3D applications.

Sharing Palettes

Palettes can be shared among multiple surfaces. The same palette can be set on the front buffer and the back buffer of a flipping chain or shared among multiple texture surfaces. When an application attaches a palette to a surface by using the **IDirectDrawSurface4::SetPalette** method, the surface increments the reference count of that palette. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached palette. In addition, if a palette is detached from a surface by using **IDirectDrawSurface4::SetPalette** with a NULL palette interface pointer, the reference count of the surface's palette will be decremented.

Note

If **IDirectDrawSurface4::SetPalette** is called several times consecutively on the same surface with the same palette, the reference count for the palette is incremented only once. Subsequent calls do not affect the palette's reference count.

Palette Animation

Palette animation refers to the process of modifying a surface's palette to change how the surface itself looks when displayed. By repeatedly changing the palette, the surface appears to change without actually modifying the contents of the surface. To this end, palette animation gives you a way to modify the appearance of a surface without changing its contents and with very little overhead.

There are two methods for providing straightforward palette animation:

- Modifying palette entries within a single palette
- Switching between multiple palettes

Using the first method, you change individual palette entries that correspond to the colors you want to animate, then reset the entries with a single call to the **IDirectDrawPalette::SetEntries** method.

The second method requires two or more **DirectDrawPalette** objects. When using this method, you perform the animation by attaching one palette object after another to the surface object by calling the **IDirectDrawSurface4::SetPalette** method.

Neither method is hardware intensive, so use whichever technique works best for your application.

For specific information and an example of how to implement palette animation, see Tutorial 5: Dynamically Modifying Palettes.

Clippers

This section contains information about **DirectDrawClipper** objects. The following topics are discussed:

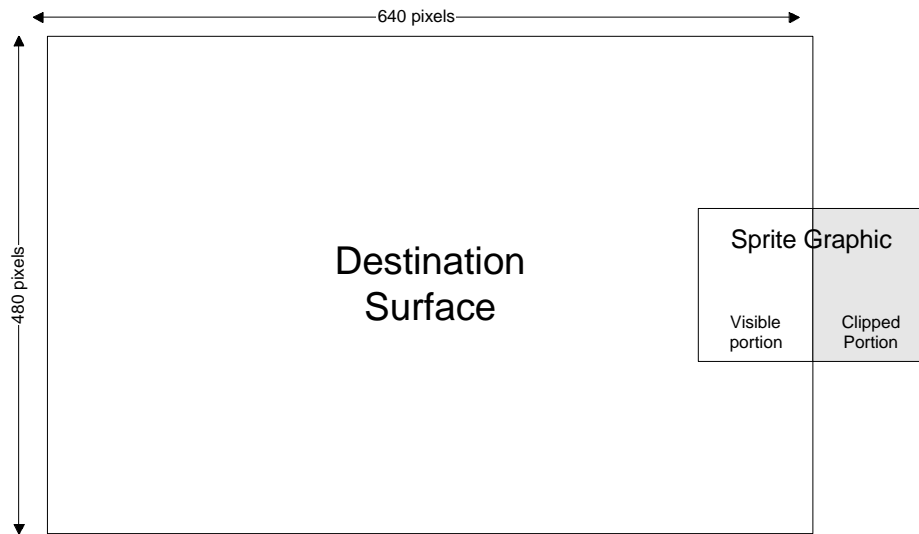
- What Are Clippers?
- Clip Lists
- Sharing **DirectDrawClipper** Objects
- Independent **DirectDrawClipper** Objects
- Creating **DirectDrawClipper** Objects with **CoCreateInstance**
- Using a Clipper with the System Cursor
- Using a Clipper with Multiple Windows

What Are Clippers?

Clippers, or **DirectDrawClipper** objects, allow you to blit to selected parts of a surface represented by a bounding rectangle or a list of several bounding rectangles. (See Clip Lists.)

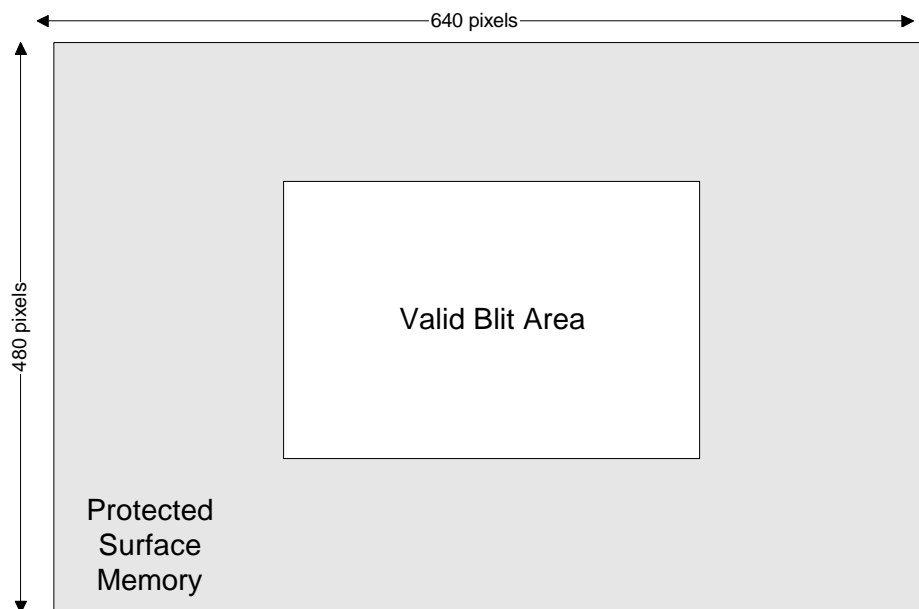
One common use for a clipper is to define the boundaries of the screen or window. For example, imagine that you want to display a sprite as it enters the screen from an edge. You don't want to make the sprite pop suddenly onto the screen; you want it to appear as though it is smoothly moving into view. Without a clipper object, **DirectDraw** does not allow you to blit the entire sprite, because part of it would fall outside the destination surface. A straight copy of the pixel values in the sprite to the destination surface buffer would result in an incorrect display and even memory access violations. With a clipper that has the screen rectangle as its clip list, **DirectDraw** knows how to trim the sprite as it performs the blit so that only the visible portion is copied.

The following illustration shows this type of clipping.



You can also use clipper objects to designate certain areas within a destination surface as writable. DirectDraw clips blit operations in these areas, protecting the pixels outside the specified clipping rectangle.

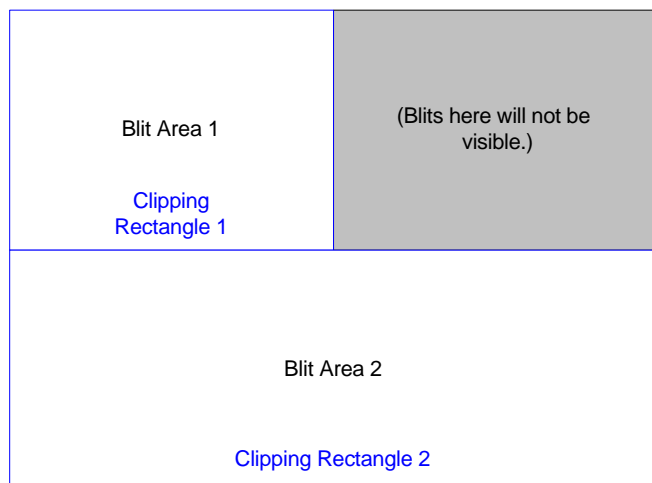
The following illustration shows this use of a clipper.



Clip Lists

A clip list consists of one or more **RECT** structures, in pixel coordinates. DirectDraw manages clip lists by using a `DirectDrawClipper` object, which can be attached to any surface.

The `IDirectDrawSurface4::Blit` method copies data only to rectangles in the clip list. For instance, if the upper-right quarter of a surface was excluded by the rectangles in the clip list, and an application blitted to the entire area of the clipped surface, DirectDraw would effectively perform two blits, the first being to the upper-left corner of the surface, and the second being to the bottom half of the surface, as shown in the following diagram.



You can manage a surface's clip list manually or, for a primary surface, have it done automatically by DirectDraw.

To manage the clip list yourself, create a list of rectangles in the form of a **RGNDATA** structure and pass this to the **IDirectDrawClipper::SetClipList** method.

To have DirectDraw manage the clip list for a primary surface, you attach the clipper to a window (even a full-screen window) by calling the **IDirectDrawClipper::SetHWND** method, specifying the target window's handle. This has the effect of setting the clipping region to the client area of the window and ensuring that the clip list is automatically updated as the window is resized, covered, or uncovered.

If you set a clipper using a window handle, you cannot set additional rectangles.

Clipping for overlay surfaces is supported only if the overlay hardware can support clipping and if destination color keying is not active.

Sharing DirectDrawClipper Objects

DirectDrawClipper objects can be shared between multiple surfaces. For example, the same DirectDrawClipper object can be set on both the front buffer and the back buffer of a flipping chain. When an application attaches a DirectDrawClipper object to a surface by using the **IDirectDrawSurface4::SetClipper** method, the surface increments the reference count of that object. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached DirectDrawClipper object. In addition, if a DirectDrawClipper object is detached from a surface by calling **IDirectDrawSurface4::SetClipper** with a NULL clipper interface pointer, the reference count of the surface's DirectDrawClipper object will be decremented.

Note

If **IDirectDrawSurface4::SetClipper** is called several times consecutively on the same surface for the same DirectDrawClipper object, the reference count for the object is incremented only once. Subsequent calls do not affect the object's reference count.

Independent DirectDrawClipper Objects

You can create DirectDrawClipper objects that are not directly owned by any particular DirectDraw object. These DirectDrawClipper objects can be shared across multiple DirectDraw objects. Driver-independent DirectDrawClipper objects are created by using the new **DirectDrawCreateClipper** DirectDraw function. An application can call this function before any DirectDraw objects are created.

Because DirectDraw objects do not own these DirectDrawClipper objects, they are not automatically released when your application's objects are released. If the application does not explicitly release these DirectDrawClipper objects, DirectDraw will release them when the application closes.

You can still create DirectDrawClipper objects by using the **IDirectDraw4::CreateClipper** method. These DirectDrawClipper objects are automatically released when the DirectDraw object from which they were created is released.

Creating DirectDrawClipper Objects with CoCreateInstance

DirectDrawClipper objects have full class-factory support for COM compliance. In addition to using the standard **DirectDrawCreateClipper** function and **IDirectDraw4::CreateClipper** method, you can also create a DirectDrawClipper object either by using the **CoGetClassObject** function to obtain a class factory and then calling the **CoCreateInstance** function, or by calling **CoCreateInstance** directly. The following example shows how to create a DirectDrawClipper object by using **CoCreateInstance** and the **IDirectDrawClipper::Initialize** method.

```
ddrval = CoCreateInstance(&CLSID_DirectDrawClipper,
    NULL, CLSCTX_ALL, &IID_IDirectDrawClipper, &lpClipper);
if (!FAILED(ddrval))
    ddrval = IDirectDrawClipper_Initialize(lpClipper,
        lpDD, 0UL);
```

In this call to **CoCreateInstance**, the first parameter, *CLSID_DirectDrawClipper*, is the class identifier of the DirectDrawClipper object class, the *IID_IDirectDrawClipper* parameter identifies the currently supported interface, and the *lpClipper* parameter points to the DirectDrawClipper object that is retrieved.

An application must use the **IDirectDrawClipper::Initialize** method to initialize DirectDrawClipper objects that were created by the class-factory mechanism before it can use the object. The value 0UL is the *dwFlags* parameter, which in this case has a value of 0 because no flags are currently supported. In the example shown here, *lpDD* is the DirectDraw object that owns the DirectDrawClipper object. However, you could supply a NULL value instead, which would create an independent DirectDrawClipper object. (This is equivalent to creating a DirectDrawClipper object by using the **DirectDrawCreateClipper** function.)

Before you close the application, close the COM library by using the **CoUninitialize** function.

Using a Clipper with the System Cursor

DirectDraw applications often need to provide a way for users to navigate using the mouse. For full-screen exclusive mode applications that use page-flipping, the only option is to implement a mouse cursor manually with a sprite, moving the sprite based on data retrieved from the device by DirectInput® or by responding to Windows mouse messages. However, any application that doesn't use page-flipping can still use the system's mouse cursor support.

When you use the system mouse cursor, you will sometimes fall victim to graphic artifacts that occur when you blit to parts of the primary surface. These artifacts appear as portions of the mouse cursor seemingly left behind by the system.

A DirectDrawClipper object can prevent these artifacts from appearing by preventing the mouse cursor image from "being in the way" during a blit operation. It's a relatively simple matter to implement, as well. To do so, create a DirectDrawClipper object by calling the **IDirectDraw4::CreateClipper** method. Then, assign your application's window handle to the clipper with the **IDirectDrawClipper::SetHWND** method. Once a clipper is attached, any subsequent blits you perform on the primary surface with the **IDirectDrawSurface4::Blt** method will not exhibit the artifact.

Note that the **IDirectDrawSurface4::BltFast** method, and its counterparts in the **IDirectDrawSurface**, **IDirectDrawSurface2**, and **IDirectDrawSurface3** interfaces, will not work on surfaces with attached clippers.

Using a Clipper with Multiple Windows

You can use a DirectDrawClipper object to blit to multiple windows created by an application running at the normal cooperative level.

To do this, create a single DirectDraw object with a primary surface. Then, create a DirectDrawClipper object and assign it to your primary surface by calling the **IDirectDrawSurface4::SetClipper** method. To blit only to the client area of a window, set the clipper to that window's client area by calling the **IDirectDrawClipper::SetHWND** method before blitting to the primary surface. Whenever you need to blit to another window's client area, call the **IDirectDrawClipper::SetHWND** method again with the new target window handle.

Creating multiple DirectDraw objects that blit to each others' primary surface is not recommended. The technique just described provides an efficient and reliable way to blit to multiple client areas with a single DirectDraw object.

Multiple Monitor Systems

Windows 98 and Windows 2000 support multiple display devices and monitors on a single system. The multiple monitor architecture (sometimes referred to as "MultiMon") enables the operating system to use the display area from two or more display devices and monitors to create a single logical desktop. For example, in a MultiMon system with two monitors, the user could display applications on either monitor, or even drag windows from one monitor to another. DirectDraw supports this architecture, allowing applications to directly access hardware on multiple display devices in a MultiMon system.

Note

As long as it is created on the null device and is not rendering directly to the primary surface, a non-full-screen DirectDraw application will work automatically with MultiMon, and the user will be able to drag the window from one monitor to another. However, DirectDraw will take advantage of hardware acceleration only when the window is entirely within the primary display. It is recommended that windowed DirectDraw applications be specifically designed for MultiMon by maintaining separate DirectDraw objects and surfaces for each monitor. For more information, see *Devices and Acceleration in MultiMon Systems*.

This section contains information about using DirectDraw on systems with multiple monitor support. The following topics are discussed:

- Enumerating Devices on MultiMon Systems
- DirectDraw Objects on Multiple Monitors
- Focus and Device Windows
- Devices and Acceleration in MultiMon Systems
- Debugging Full-Screen DirectDraw Applications with MultiMon

The `Multimon.h` header file included with the DirectX Programmer's Reference makes it possible for code written around Windows 98 multiple monitor functions to compile and run successfully on operating systems that do not support MultiMon.

The following sample applications demonstrate the implementation of MultiMon in DirectDraw:

- Stretch2 Sample
- Stretch3 Sample
- Multimonitor Space Donuts Sample

Enumerating Devices on MultiMon Systems

Use the **DirectDrawEnumerateEx** function to enumerate devices on systems with multiple monitors, specifying flags to determine what types of DirectDraw devices should be enumerated. The function calls an application-defined **DDEnumCallbackEx** function for each enumerated device.

The **DirectDrawEnumerateEx** function is supported on Windows 98 and Windows 2000 operating systems. It is available in `Ddraw.lib` for applications compiled under DirectX 6.0 and later versions. Applications that statically link to the function will always run under DirectX 6.0 and later, and will always run under any version of DirectX on Windows 98 and Windows 2000. Such applications will fail if run on previous versions of DirectX under Windows 95.

If your application needs to run on versions of DirectX older than DirectX 5.0, it should use **GetProcAddress** to see if **DirectDrawEnumerateEx** is available. The following example shows one way you can do this:

```
HINSTANCE h = LoadLibrary("ddraw.dll");

// If ddraw.dll doesn't exist in the search path,
// then DirectX probably isn't installed, so fail.
if (!h)
    return FALSE;

// Note that you must know which version of the
// function to retrieve (see the following text).
// For this example, we use the ANSI version.
LPDIRECTDRAWENUMERATEEX lpDDEnumEx;
lpDDEnumEx = (LPDIRECTDRAWENUMERATEEX) GetProcAddress(h, "DirectDrawEnumerateExA");
```

```

// If the function is there, call it to enumerate all display
// devices attached to the desktop, and any non-display DirectDraw
// devices.
if (lpDDEnumEx)
    lpDDEnumEx(Callback, NULL,
        DDENUM_ATTACHEDSECONDARYDEVICES |
        DDENUM_NONDISPLAYDEVICES
    );
else
{
    /*
    * We must be running on an old version of DirectDraw.
    * Therefore MultiMon isn't supported. Fall back on
    * DirectDrawEnumerate to enumerate standard devices on a
    * single-monitor system.
    */
    DirectDrawEnumerate(OldCallback, NULL);

    /* Note that it could be handy to let the OldCallback function
    * be a wrapper for a DDEnumCallbackEx.
    *
    * Such a function would look like:
    *   BOOL FAR PASCAL OldCallback(GUID FAR *lpGUID,
    *                               LPSTR pDesc,
    *                               LPSTR pName,
    *                               LPVOID pContext)
    *   {
    *       return Callback(lpGUID, pDesc, pName, pContext, NULL);
    *   }
    */
}

// If the library was loaded by calling LoadLibrary(),
// then you must use FreeLibrary() to let go of it.
FreeLibrary(h);

```

The previous example will work for applications that link to Ddraw.dll at run-time or load-time.

Note that you must retrieve the address of either the ANSI or Unicode version of the **DirectDrawEnumerateEx** function, depending of the type of strings your application uses. When declaring the corresponding callback function, use the **LPTSTR** data type for the string parameters. The **LPTSTR** data type compiles to use Unicode strings if you declare the **_UNICODE** symbol, and ANSI strings otherwise. By using the **LPTSTR** data type, the function should compile properly regardless of the string type you use in your application.

DirectDraw Objects on Multiple Monitors

Windowed DirectDraw applications written for the null or default display driver will work on MultiMon systems, but in applications optimized for MultiMon you will want to create a separate DirectDraw object for each device, using the GUID returned in the enumeration callback. (See Enumerating Devices on MultiMon Systems.)

Avoid setting the cooperative level multiple times on a MultiMon system. If you need to switch from full-screen to normal mode, it is best to create a new DirectDraw object.

It is good practice to release all DirectDraw objects at the same time. If you release only the secondary device or devices, the primary device goes back to its original desktop mode, but only the taskbar is redrawn and the DirectDraw primary surface is still present. You cannot draw to this surface without first releasing the DirectDraw object and then re-creating it.

Focus and Device Windows

Each DirectDraw application that uses one or more monitors in full-screen exclusive mode must have a single focus window, which is the window that receives keyboard input.

Each device that is to hold a full-screen DirectDraw surface must be represented by a DirectDraw object and a device window. The device window is the one that is sized to fit the window and is put on top of all other windows.

For single-monitor applications, there is no distinction between the device and focus window. They are one and the same. For multiple-monitor applications, however, you need to set a device window for each monitor, and you have to let each DirectDraw object know about the application's focus window. The focus window can also serve as the device window for one of the monitors. Other device windows should be children of the focus window so that the application does not minimize when the user clicks on one of them.

See also:

- Setting the Focus Window
- Setting Device Windows

Setting the Focus Window

To set the focus window, you call the **IDirectDraw4::SetCooperativeLevel** method for each of the DirectDraw objects. You pass in a window handle (normally the application window handle) and set the `DDSCL_SETFOCUSWINDOW` flag, as in the following example:

```
/* It is presumed that lpDD is a valid IDirectDraw interface pointer,
   and that hWnd is a valid window handle. */

HRESULT ddrval = lpDD->SetCooperativeLevel( hWnd,
      DDSCL_SETFOCUSWINDOW );
```

The focus window must be the same for all devices.

Setting Device Windows

There are two ways to set a device window:

- Create a window yourself and pass its handle to the **IDirectDraw4::SetCooperativeLevel** method of the DirectDraw object representing the monitor, setting the `DDSCL_SETDEVICEWINDOW`, `DDSCL_FULLSCREEN`, and `DDSCL_EXCLUSIVE` flags. This creates a full-screen window and sets it as the device window for the monitor. Your application will receive mouse messages for the window, and you are responsible for destroying the window at the appropriate time. The window you pass to **SetCooperativeLevel** should be either the focus window (possible only if it is on the same device) or a child of the focus window.
- Let DirectDraw create the window. You pass the focus window handle to **SetCooperativeLevel** and set the `DDSCL_CREATEDEVICEWINDOW`, `DDSCL_FULLSCREEN`, and `DDSCL_EXCLUSIVE` flags. DirectDraw creates a default device window that is a child of the focus window. It manages this window and will destroy it at the appropriate time. Your application will not receive any mouse messages for the window.

The following example sets an existing device window for the DirectDraw object represented by *lpDD*.

```
/* It is presumed that lpDD is a valid IDirectDraw interface pointer,
   and that hWnd is the handle to an appropriate device window. */

HRESULT hr = lpDD->SetCooperativeLevel( hWnd,
      DDSCL_SETDEVICEWINDOW | DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
```

The following example sets a default device window created by DirectDraw. In this case, *hWnd* is the handle to the existing focus window.

```
HRESULT hr = lpDD->SetCooperativeLevel( hWnd,
      DDSCL_CREATEDEVICEWINDOW | DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
```

Although a focus window can be a device window, you cannot set a window as both the focus window and a device window with a single call to **SetCooperativeLevel**. You must first set it as the focus window and then set it as a device window. However, it is possible to set a focus window and a default device window on the same device with a single call to **SetCooperativeLevel**. The following example shows how this can be done:

```
HRESULT hr = lpDD->SetCooperativeLevel (
    hwndFocus,
    DDSCL_SETFOCUSWINDOW | DDSCL_FULLSCREEN |
    DDSCL_EXCLUSIVE | DDSCL_CREATEDEVICEWINDOW);
```

In this example, an existing window (probably the application window) is set as the focus window, and DirectDraw creates a default device window.

Devices and Acceleration in MultiMon Systems

Full-screen exclusive mode DirectDraw objects will take advantage of hardware acceleration regardless of whether they are running on the primary device or on a secondary device. However, they cannot use built-in support for spanning graphics operations across display devices. It is the application's responsibility to perform operations on the appropriate device.

When the normal cooperative level is set, DirectDraw uses hardware acceleration only when the window is wholly within the display area of the primary device. When a window straddles two or more monitors, all blits are done in emulation and performance can be significantly slower. This is necessarily the case, because hardware buffers cannot blit to a display surface controlled by another piece of hardware.

As long as you create the DirectDraw object for the null device—that is, pass NULL to **DirectDrawCreate** as the *lpGUID* parameter—DirectDraw will blit to the entire window regardless of where it is located. However, if the device is created by its actual GUID, this is not the case, and blit operations that cross an edge of the primary surface will be clipped (if you are using a clipper) or will fail, returning DDERR_INVALIDRECT.

Note

When you are blitting to a window in a MultiMon application, negative coordinates are valid when the logical location of the secondary monitor is to the left of the primary monitor.

To get the best performance in a windowed MultiMon application, you need to create a DirectDraw object for each device, maintain off-screen surfaces in parallel on each device, keep track of which part of the window resides on each device, and perform separate blits to each device.

Debugging Full-Screen DirectDraw Applications with MultiMon

It is possible to use a multimonitor system rather than remote debugging in order to step through code while debugging a full-screen DirectDraw application.

You should use the primary monitor for your development environment and the secondary monitor for the DirectDraw output. Also, you need to change a registry setting through the DirectX property sheet in Control Panel. On the **DirectDraw** page, click **Advanced Settings** and select the **Enable Multi-Monitor Debugging** checkbox. This setting will prevent DirectDraw from minimizing your application when it loses focus.

Under Windows 98, you cannot step through code when a surface is locked. For more information, see [Accessing Surface Memory Directly](#).

Advanced DirectDraw Topics

This section supplements the DirectDraw overview, providing information about advanced DirectDraw issues. The following topics are discussed:

- Mode 13 Support
- Taking Advantage of DMA Support
- Using DirectDraw Palettes in Windowed Mode
- Video Ports
- Getting the Flip and Blit Status

- Determining the Capabilities of the Display Hardware
- Storing Bitmaps in Display Memory
- Triple Buffering
- DirectDraw Applications and Window Styles
- Matching True RGB Colors to the Frame Buffer's Color Space
- Displaying a Window in Full-Screen Mode

Mode 13 Support

This section contains information about DirectDraw Mode 13 graphics mode support. The following topics are discussed:

- About Mode 13
- Setting Mode 13
- Mode 13 and Surface Capabilities
- Using Mode 13

About Mode 13

DirectDraw supports access to the linear unflippable 320x200 8 bits per pixel palettized mode known widely by the name Mode 13, its hexadecimal BIOS mode number. DirectDraw treats this mode like a Mode X mode, but with some important differences imposed by the physical nature of Mode 13.

Setting Mode 13

Mode 13 has similar enumeration and mode-setting behavior as Mode X. DirectDraw will only enumerate Mode 13 if the `DDSCCL_ALLOWMODEX` flag was passed to the **IDirectDraw4::SetCooperativeLevel** method.

You enumerate the Mode 13 display mode like all other modes, but you make a surface capabilities check before calling **IDirectDraw4::EnumDisplayModes**. To do this, call **IDirectDraw4::GetCaps** and check for the `DDSCAPS_STANDARDVGAMODE` flag in the **DDSCAPS2** structure after the method returns. If this flag is not present, then Mode 13 is not supported, and attempts to enumerate with the `DDEDM_STANDARDVGAMODES` flag will fail, returning `DDERR_INVALIDPARAMS`.

The **EnumDisplayModes** method now supports a new enumeration flag, `DDEDM_STANDARDVGAMODES`, which causes DirectDraw to enumerate Mode 13 in addition to the 320x200x8 Mode X mode. There is also a new **IDirectDraw4::SetDisplayMode** flag, `DDSDM_STANDARDVGAMODE`, which you must pass in order to distinguish Mode 13 from 320x200x8 Mode X.

Note that some video cards offer linear accelerated 320x200x8 modes. On such cards DirectDraw will not enumerate Mode 13, enumerating the linear mode instead. In this case, if you attempt to set Mode 13 by passing the `DDSDM_STANDARDVGAMODE` flag to **SetDisplayMode**, the method will succeed, but the linear mode will be used. This is analogous to the way that linear low resolution modes override Mode X modes.

Mode 13 and Surface Capabilities

When DirectDraw calls an application's **EnumModesCallback** callback function, the **ddsCaps** member of the associated **DDSURFACEDESC** or **DDSURFACEDESC2** structure contains flags that reflect the mode being enumerated. You can expect `DDSCAPS_MODEX` for a Mode X mode or `DDSCAPS_STANDARDVGAMODE` for Mode 13. These flags are mutually exclusive. If neither of these bits is set, then the mode is a linear accelerated mode. This behavior also applies to the flags retrieved by the **IDirectDraw4::GetDisplayMode** method.

Using Mode 13

Because Mode 13 is a linear mode, unlike the Mode X modes, DirectDraw can give an application direct access to the frame buffer. You can call the **IDirectDrawSurface4::Lock**, **IDirectDrawSurface4::Blt**, and **IDirectDrawSurface4::BltFast** methods to gain direct access to the primary surface.

When using Mode 13, DirectDraw supports an emulated **IDirectDrawSurface4::Flip** that is implemented as a straight copy of the contents of a back buffer to the primary surface. You can emulate this yourself by copying all or part of the back-buffer's contents to the primary surface using **Blt** or **BltFast**.

There is one warning concerning **Lock** and Mode 13. Although DirectDraw allows direct linear access to the Mode 13 VGA frame buffer, do not assume that the buffer is always located at address 0xA0000, since DirectDraw can return an aliased virtual-memory pointer to the frame buffer which will not be 0xA0000. Similarly, do not assume that the pitch of a Mode 13 surface is 320, because display cards that support an accelerated 320x200x8 mode will very likely use a different pitch.

Taking Advantage of DMA Support

This section contains information about how you can take advantage of device support for Direct Memory Access (DMA) to increase performance in completing certain tasks. The following topics are discussed:

- About DMA Device Support
- Testing for DMA Support
- Typical Scenarios for DMA
- Using DMA

About DMA Device Support

Some display devices can perform blit operations (or other operations) on system memory surfaces. These operations are commonly referred to as Direct Memory Access (DMA) operations. You can exploit DMA support to accelerate certain combinations of operations. For example, on such a device, you could perform a blit from system memory to video memory while using the processor to prepare the next frame. In order to use such facilities, you must assume certain responsibilities. This section details these tasks.

Testing for DMA Support

Before using DMA operations, you must test the device for DMA support and, if it does support DMA, how much support it provides. Begin by retrieving the driver capabilities by calling the **IDirectDraw4::GetCaps** method, then look for the **DDCAPS_CANBLTSYSMEM** flag in the **dwCaps** member of the associated **DDCAPS** structure. If the flag is set, the device supports DMA.

If you know that DMA is generally supported, you also need to find out how well the driver supports it. You do so by looking at some other structure members that provide information about system-to-video, video-to-system, and system-to-system blit operations. These capabilities are provided in 12 **DDCAPS** structure members that are named according to blit and capability type. The following table shows these new members.

System-to-video	Video-to-system	System-to-system
dwSVBCaps	dwVSBCaps	dwSSBCaps
dwSVBCKeyCaps	dwVSBCKeyCaps	dwSSBCKeyCaps
dwSVBFXCaps	dwVSBFXCaps	dwSSBFXCaps
dwSVBRops	dwVSBRops	dwSSBRops

For example, the system-to-video blit capability flags are provided in the **dwSVBCaps**, **dwSVBCKeyCaps**, **dwSVBFXCaps** and **dwSVBRops** members. Similarly, video-to-system blit capabilities are in the members whose names begin with "**dwVSB**," and system-to-system capabilities are in the "**dwSSB**" members. Examine the flags present in these members to determine the level of hardware support for that blit category.

The flags in these members are parallel with the blit-related flags included in the **dwCaps**, **dwCKeyCaps**, and **dwFXCaps** members, with respect to that member's blit type. For example, the **dwSVBCaps** member contains general blit capabilities as specified by the same flags you might find in the **dwCaps** member. Likewise, the raster operation values in the **dwSVBRops**, **dwVSBRops**, and **dwSSBRops** members provide information about the raster operations supported for a given type of blit operation.

One of the key features to look for in these members is support for asynchronous DMA blit operations. If the driver supports asynchronous DMA blits between surfaces, the **DDCAPS_BLTQUEUE** flag will be set in the **dwSVBCaps**, **dwVSBCaps**, or **dwSSBCaps** member. (Generally, you'll see the best support for system-memory-to-video-memory surfaces.) If the flag isn't present, the driver isn't reporting support for asynchronous DMA blit operations.