Chapter 2

Microsoft® DirectX™ 3 Software Development Kit

DirectDraw

DirectDraw

CHAPTER 2

About DirectDraw

DirectDraw® is a DirectX™ SDK component that allows direct manipulation of display memory, hardware blitters, hardware overlays, and flipping. 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 3D 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 the following benefits that were previously available only to applications that included code for specific display devices:

- Support for double-buffered and flipping graphics
- Access to, and control of, the display card's blitter
- Support for 3D 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

DirectDraw's mission is to provide 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 worry about the specific calling procedures required to utilize 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 Windows NT® version 4.0 or later.

DirectDraw Architecture

DirectDraw provides display-memory and display-hardware management services. It also provides the usual functionality associated with memory management: Memory can be allocated, moved, transformed, and freed. This memory represents visual images and is referred to as a *surface*. Through the DirectDraw hardware-abstraction layer (HAL), applications are exposed to unique display hardware functionality, including stretching, overlaying, texture mapping, rotating, and mirroring.

DirectDraw Overview

This section contains general information about the DirectDraw component. The following topics are discussed:

- *DirectDraw*
- *Other DirectDraw Features*
- *DirectDraw HAL*
- *DirectDraw HEL*
- *Types of DirectDraw Objects*
- *Width and Pitch*
- *Support for 3D Surfaces*
- *Direct3D Integration with DirectDraw*
- *Mode X Display Mode*
- *Pixel Formats*

DirectDraw

DirectDraw is implemented in both hardware and software. Applications using DirectDraw retrieve two sets of capabilities, one for hardware capabilities and one for software-emulation capabilities. Using these, an application can easily determine what DirectDraw is emulating and what functionality is provided in hardware. DirectDraw is the only client of the DirectDraw hardware-abstraction layer (HAL). Applications must write to DirectDraw; there is no mechanism for writing to the HAL more directly.

DirectDraw is implemented by the Ddraw dynamic-link library (DLL). This 32-bit DLL implements all of the common functionality required by DirectDraw. It performs all of the necessary thunking between Win32® and the 16-bit portions of the HAL, as well as complete parameter validation. It provides management for off-screen display memory, and performs all of the bookkeeping and semantic logic required for DirectDraw. It is responsible for presenting the Component

Object Model (COM) interface to the application, hooking window handles to provide clip lists, and all other device-independent functionality.

DirectDraw is a combination of four COM interfaces: *IDirectDraw2*, *IDirectDrawSurface2*, *IDirectDrawPalette*, and *IDirectDrawClipper*. For information about COM concepts, see *The Component Object Model*.

The first two objects a DirectDraw application uses are DirectDraw and DirectDrawSurface. A DirectDraw object, created by using the **DirectDrawCreate** function, represents the display adapter card. After retrieving an *IDirectDraw2* interface to the object, an application can call the **IDirectDraw2::CreateSurface** method to create the primary DirectDrawSurface object, which represents the display memory being viewed on the monitor. From the primary surface, off-screen surfaces can be created in a linked-list fashion.

In the most common case, one back buffer is created to exchange images with the primary surface. While the screen is busy displaying the lines of the image in the primary surface, the back-buffer surface frame is composed. This composition is performed by transferring to the back buffer a series of off-screen bitmaps stored on other DirectDrawSurface objects in display memory. To display the recently composed frame, the application calls the **IDirectDrawSurface2::Flip** method. This method sets a register so that the exchange occurs when the screen performs a vertical retrace. This operation is asynchronous, so the application can continue processing after calling this method. (After this method has been called, the back buffer is automatically write-blocked until the exchange occurs.) After the exchange occurs, this process continues: The application composes the next frame in the back buffer, calls **IDirectDrawSurface2::Flip**, and so on.

DirectDraw improves performance over the Windows version 3.1 GDI model, which had no direct access to bitmaps in display memory. Blits, which always occurred in system memory, were subequently transferred to display memory, slowing performance. With DirectDraw, processing is done on the display adapter card whenever possible. DirectDraw also improves performance over the Windows 95 and Windows NT GDI model, which uses the **CreateDIBSection** function to enable hardware processing.

A third DirectDraw object is DirectDrawPalette. Because the physical display palette is usually maintained in display hardware, an object represents and manipulates it. The **IDirectDrawPalette** interface implements palettes in hardware. These bypass Windows palettes and are therefore available only when a game has exclusive access to the display hardware. DirectDrawPalette objects are also created from DirectDraw objects.

The final DirectDraw object is DirectDrawClipper. DirectDraw manages clipped regions of display memory by using this object.

Other DirectDraw Features

In addition to the features discussed in *DirectDraw*, DirectDraw also supports *transparent blitting* and *overlays*.

In transparent blitting, a bitmap is transferred to a surface and a certain color, or range of colors, in the bitmap is defined as transparent. Transparent blits are achieved by using *color keying*. *Source* color keying operates by defining which color or color range on the bitmap is transparent and therefore not copied during a transfer operation. *Destination* color keying operates by defining which color or color range on the surface will be covered by pixels of that color or color range in the source bitmap. For more information about color keying, see *Color Keying*.

Finally, DirectDraw supports overlays in hardware and by software emulation. Overlays present an easier means of implementing sprites and managing multiple layers of animation. Any DirectDrawSurface object can be created as an overlay with all of the capabilities of any other surface, in addition to the extra capabilities associated only with overlays. These capabilities require extra display memory. If there are no overlays in display memory, the overlay surfaces can exist in system memory.

Color keying works in the same way for overlays as for transparent blits. The z-order of the overlay automatically handles the occlusion and transparency manipulations between overlays.

DirectDraw HAL

The DirectDraw hardware-abstraction layer (HAL) is hardware-dependent and contains only hardware-specific code. The HAL can be implemented in 16 bits, 32 bits, or, on Windows 95, a combination of the two. The HAL is always implemented in 32 bits on Windows NT. The HAL can be an integral part of the display driver or a separate DLL that communicates with the display driver through a private interface defined by the driver's creator.

The DirectDraw HAL is implemented by the chip manufacturer, board producer, or original equipment manufacturer (OEM). The HAL implements only the 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. The HAL also does no parameter validation; parameters are validated by DirectDraw before the HAL is invoked.

DirectDraw HEL

DirectDraw's hardware-emulation layer (HEL) presents its capabilities to DirectDraw just as a HAL would. By examining these capabilities during application initialization, you can adjust application parameters to provide optimum performance on a variety of platforms. If a DirectDraw HAL is not

present or a requested feature is not provided by the hardware, DirectDraw will emulate the missing functionality.

Types of DirectDraw Objects

The DirectDraw object represents the display device. Multiple DirectDraw objects can be created for each logical display device. Each unique DirectDraw object can create surface, palette, and clipper objects that are independent of all other DirectDraw objects.

A DirectDrawSurface object represents a linear region of display memory that can be directly accessed and manipulated. These display memory addresses may point to visible frame buffer memory (primary surface) or to non-visible buffers (off-screen or overlay surfaces). These non-visible buffers usually reside in display memory, but they can be created in system memory if required by the hardware design or if DirectDraw is performing software emulation.

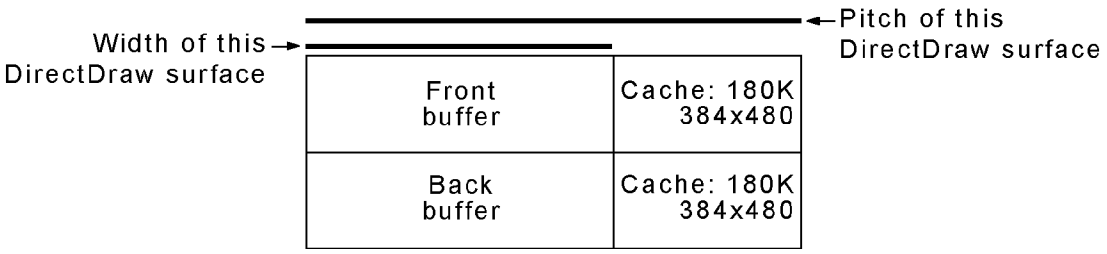
A DirectDrawPalette object represents either a 16- or a 256-color indexed palette. Palettes are provided for textures, off-screen surfaces, and overlay surfaces, none of which are required to have the same palette as the primary surface.

The DirectDraw object creates DirectDrawSurface, DirectDrawPalette, and DirectDrawClipper objects. DirectDrawPalette and DirectDrawClipper objects must be attached to the DirectDrawSurface objects they affect. A DirectDrawSurface object may refuse the request to attach a DirectDrawPalette object to it. This is not unusual, because most hardware does not support multiple palettes.

Width and Pitch

If your application writes to display memory, bitmaps stored in memory do not necessarily occupy a contiguous block of memory. In this case, the *width* and *pitch* of a line in a bitmap can be different. The width is the distance between two addresses in memory that represent the beginning of a line and the end of the line of a stored bitmap. This distance represents only the width of the bitmap in memory; it does not include any extra memory required to reach the beginning of the next line of the bitmap. The pitch is the distance between two addresses in memory that represent the beginning of a line and the beginning of the next line in a stored bitmap.

For rectangular memory, for example, the pitch of the display memory will include the width of the bitmap plus part of a cache. The following figure shows the difference between width and pitch in rectangular memory:



In this figure, the front buffer and back buffer are both 640×480×8, and the cache is 384×480×8. To reach the address of the next line to write to the buffer, you must add 640 and 384 to get 1024, which is the beginning of the next line.

Therefore, when rendering directly into surface memory, always use the pitch returned by the **IDirectDrawSurface2::Lock** method (or the **IDirectDrawSurface2::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.

Support for 3D Surfaces

This section contains information about DirectDraw's 3D-surface capabilities. The following topics are discussed:

- *Texture Maps*
- *Mipmaps*
- *Z-Buffers*

Texture Maps

Texture maps can be allocated in system memory by using the HEL. To allocate a texture map surface, specify the **DDSCAPS_TEXTURE** flag in the **ddsCaps** member of the **DDSURFACEDESC** structure passed to the **IDirectDraw2::CreateSurface** method.

A wide range of texture pixel formats is supported by the HEL. For a list of these formats, see *Texture Map Formats*.

Mipmaps

DirectDraw supports *mipmapped* texture surfaces. A mipmap is a sequence of textures, each of which is the same image at a progressively lower resolution. Each prefiltered image, or level, in the mipmap is a power of two smaller than the previous level. A high-resolution level is used for objects that are close to the viewer. Lower-resolution levels are used as the object moves farther away.

Mipmapping is a computationally low-cost way of improving the quality of rendered textures.

In DirectDraw, mipmaps are represented as a chain of attached surfaces. The highest resolution texture is at the head of the chain and has, as an attachment, the next level of the mipmap. That level has, in turn, an attachment that is the next level in the mipmap, and so on down to the lowest resolution level of the mipmap.

To create a surface representing a single level of a mipmap, specify the **DDSCAPS_MIPMAP** flag in the **DDSURFACEDESC** structure passed to the **IDirectDraw2::CreateSurface** method. Because all mipmaps are also textures, the **DDSCAPS_TEXTURE** flag must also be specified. It is possible to create each level manually and build the chain by using the **IDirectDrawSurface2::AddAttachedSurface** method. However, you can use the **IDirectDraw2::CreateSurface** method to build an entire mipmap chain in a single operation.

The following example demonstrates building a chain of five mipmap levels of sizes 256×256, 128×128, 64×64, 32×32, and 16×16:

```
DDSURFACEDESC       ddsd;
LPDIRECTDRAWSURFACE2 lpDDMipMap;
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_MIPMAPCOUNT;
ddsd.dwMipMapCount = 5;
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE |
    DDSCAPS_MIPMAP | DDSCAPS_COMPLEX;
ddsd.dwWidth = 256UL;
ddsd.dwHeight = 256UL;

ddres = lpDD->CreateSurface(&ddsd, &lpDDMipMap);
if (FAILED(ddres))
.
.
.
```

You can omit the number of mipmap levels, in which case the **IDirectDraw2::CreateSurface** method will create a chain of surfaces, each a power of two smaller than the previous one, down to the smallest possible size. It is also possible to omit the width and height, in which case **IDirectDraw2::CreateSurface** will create the number of levels you specify, with a minimum level size of 1×1.

A chain of mipmap surfaces is traversed by using the **IDirectDrawSurface2::GetAttachedSurface** method and specifying the **DDSCAPS_MIPMAP** and **DDSCAPS_TEXTURE** flags in the **DDSCAPS** structure. The following example traverses a mipmap chain from highest to lowest resolutions:

Note

```
LPDIRECTDRAWSURFACE lpDDLevel, lpDDNextLevel;
DDSCAPS ddsCaps;

lpDDLevel = lpDDMipMap;
lpDDLevel->AddRef();
ddsCaps.dwCaps = DDSCAPS_TEXTURE | DDSCAPS_MIPMAP;
ddres = DD_OK;
while (ddres == DD_OK)
{
    // Process this level.
    .
    .
    .
    ddres = lpDDLevel->GetAttachedSurface(
        &ddsCaps, &lpDDNextLevel);
    lpDDLevel->Release();
    lpDDLevel = lpDDNextLevel;
}
if ((ddres != DD_OK) && (ddres != DDERR_NOTFOUND))
.
.
.
```

You can also build flipping chains of mipmaps. In this scenario, each mipmap level has an associated chain of back buffer texture surfaces. Each back-buffer texture surface is attached to one level of the mipmap. Only the front buffer in the chain has the **DDSCAPS_MIPMAP** flag set; the others are simply texture maps (created by using the **DDSCAPS_TEXTURE** flag). A mipmap level can have two attached texture maps, one with **DDSCAPS_MIPMAP** set, which is the next level in the mipmap chain, and one with the **DDSCAPS_BACKBUFFER** flag set, which is the back buffer of the flipping chain. All the surfaces in each flipping chain must be of the same size.

It is not possible to build such a surface arrangement with a single call to the **IDirectDraw2::CreateSurface** method. To construct a flipping mipmap, either build a complex mipmap chain and manually attach back buffers by using the **IDirectDrawSurface2::AddAttachedSurface** method, or create a sequence of flipping chains and build the mipmap by using **IDirectDrawSurface2::AddAttachedSurface**.

Blit operations apply to only a single level in the mipmap chain. To blit an entire chain of mipmaps, each level must be blitted separately.

The **IDirectDrawSurface2::Flip** method will flip all the levels of a mipmap from the level supplied to the lowest level in the mipmap. A destination surface can also be provided, in which case all levels in the mipmap will flip to the back buffer in their flipping chain. This back buffer matches the supplied override. For

Note

example, if the third back buffer in the top-level flipping chain is supplied as the override, all levels in the mipmap will flip to the third back buffer.

The number of levels in a mipmap chain is stored explicitly. When an application obtains the surface description of a mipmap (by calling the **IDirectDrawSurface2::Lock** or **IDirectDrawSurface2::GetSurfaceDesc** method), the **dwMipMapCount** member of the **DDSURFACEDESC** structure will contain the number of levels in the mipmap, including the top level. For levels other than the top level in the mipmap, the **dwMipMapCount** member specifies the number of levels from that mipmap to the smallest mipmap in the chain.

Z-Buffers

The DirectDraw HEL can create z-buffers for use by Direct3D™ or other 3D-rendering software. The HEL supports both 16- and 32-bit z-buffers. The DirectDraw device driver for a 3D-accelerated display card can permit the creation of z-buffers in display memory by exporting the **DDSCAPS_ZBUFFER** flag. It should also specify the z-buffer depths it supports by using the **dwZBufferBitDepths** member of the **DDCAPS** structure.

An application can clear z-buffers by using the **IDirectDrawSurface2::Blt** method. The **DDBLT_DEPTHFILL** flag indicates that the blit clears z-buffers. If this flag is specified, the **DDBLTFX** structure passed to the **IDirectDrawSurface2::Blt** method should have its **dwFillDepth** member set to the required z-depth. If the DirectDraw device driver for a 3D-accelerated display card is designed to provide support for z-buffer clearing in hardware, it should export the **DDCAPS_BLTDEPTHFILL** flag and should handle **DDBLT_DEPTHFILL** blits. The destination surface of a depth-fill blit must be a z-buffer.

The actual interpretation of a depth value is specific to the 3D renderer.

Direct3D Integration with DirectDraw

This section describes the relationship between DirectDraw and Direct3D. The following topics are discussed:

- *Direct3D Driver Interface*
- *Direct3D Device Interface*
- *Direct3D Texture Interface*
- *DirectDraw HEL and Direct3D*

Direct3D Driver Interface

DirectDraw presents programmers with a single, unified object. This object encapsulates both the DirectDraw and Direct3D states. Both the DirectDraw driver COM interfaces and the Direct3D driver COM interface allow you to communicate with the same underlying object. When an application uses Direct3D, no Direct3D object is created—rather, the application uses the standard COM **QueryInterface** method to obtain a Direct3D interface to a DirectDraw object.

The following example demonstrates how to create the DirectDraw object and obtain a Direct3D interface for communicating with that object:

```
LPDIRECTDRAW lpDD;
LPDIRECT3D lpD3D;
ddres = DirectDrawCreate(NULL, &lpDD, NULL);
if (FAILED(ddres))
.
.
.
ddres = lpDD->QueryInterface(IID_IDirect3D,
    &lpD3D);
if (FAILED(ddres))
.
.
.
```

The code shown in the previous example creates a single object and obtains two interfaces to that object. Therefore, the object's reference count is 2 after the **IDirectDraw2::QueryInterface** method call. The important implication of this is that the lifetime of the Direct3D driver state is the same as that of the DirectDraw object. Releasing the Direct3D interface does not destroy the Direct3D driver state. That state is not destroyed until all references to that object —whether they are DirectDraw or Direct3D references—have been released. Therefore, if you release a Direct3D interface while holding a reference to a DirectDraw driver interface, and then query the Direct3D interface again, the Direct3D state will be preserved.

Direct3D Device Interface

As with the Direct3D object, there is no distinct Direct3D device object. A Direct3D device is simply an interface for communicating with a DirectDrawSurface object used as a 3D-rendering target. The following example creates a Direct3D device interface to a DirectDrawSurface object:

```
LPDIRECTDRAWSURFACE lpDDSurface;
LPDIRECT3DDEVICE    lpD3DDevice;

ddres = lpDD->CreateSurface(&ddsd, &lpDDSurface,
```

```
    NULL);
if (FAILED(ddres))
    .
    .
    .
ddres = lpDDSurface->QueryInterface(lpGuid,
    &lpD3DDevice);
if (FAILED(ddres))
    .
    .
    .
```

The same rules for reference counts and state lifetimes for objects apply to DirectDrawSurface objects and Direct3D devices. (For information about these rules, see *Direct3D Driver Interface*.) Additionally, multiple, distinct Direct3D device interfaces can be obtained for the same DirectDrawSurface object. It is possible, therefore, that a single DirectDrawSurface object could be the target for both a ramp-based device and an RGB-based device.

Direct3D Texture Interface

Direct3D textures are not distinct object types, but rather another interface of DirectDrawSurface objects. The following example obtains a Direct3D texture interface from a DirectDrawSurface object:

```
LPDIRECTDRAWSURFACE lpDDSurface;
LPDIRECT3DTEXTURE   lpD3DTexture;

ddres = lpDD->CreateSurface(&ddsd, &lpDDSurface,
    NULL);
if (FAILED(ddres))
    .
    .
    .
ddres = lpDDSurface->QueryInterface(
    IID_IDirect3DTexture, &lpD3DTexture);
if (FAILED(ddres))
    .
    .
    .
```

The same rules for reference counts and state lifetimes for objects apply to Direct3D textures. (For information about these rules, see *Direct3D Driver Interface*.) It is possible to use a single DirectDrawSurface object as both a rendering target and a texture.

DirectDraw HEL and Direct3D

The DirectDraw HEL supports the creation of texture, mipmap, and z-buffer surfaces. Furthermore, because of the tight integration of DirectDraw and Direct3D, a DirectDraw-enabled system always provides Direct3D support (in software emulation, at least). Therefore, the DirectDraw HEL exports the **DDSCAPS_3DDEVICE** flag to indicate that a surface can be used for 3D rendering. DirectDraw drivers for hardware-accelerated 3D display cards export this capability to indicate the presence of hardware-accelerated 3D.

Mode X Display Mode

Mode X is a hybrid display mode derived from the standard VGA Mode 13. This mode allows the use of up to 256K bytes of display memory (rather than the 64K bytes allowed by Mode 13) by using the VGA display adapter's EGA multiple video plane system. On Windows 95, 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 not currently supported on Windows NT. Linear low-resolution modes are also largely unsupported.

Pixel Formats

This section contains information about the pixel formats supported by the hardware-emulation layer (HEL). The following topics are discussed:

- *Texture Map Formats*
- *Off-Screen Surface Formats*

Texture Map Formats

A wide range of texture pixel formats are supported by the HEL. The following table shows these formats. The Masks column contains the red, green, blue, and alpha masks for each set of pixel format flags and bit depths.

Pixel format flags	Bit depth	Masks
DDPF_RGB |	1	R: 0x00000000
DDPF_PALETTEINDEXED1		G: 0x00000000
		B: 0x00000000
		A: 0x00000000
DDPF_RGB |	1	R: 0x00000000
DDPF_PALETTEINDEXED1 |		G: 0x00000000

DDPF_PALETTEINDEXEDTO8		B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2 | DDPF_PALETTEINDEXEDTO8	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4 | DDPF_PALETTEINDEXEDTO8	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED8	8	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB	8	R: 0x000000E0 G: 0x0000001C B: 0x00000003 A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	16	R: 0x00000F00 G: 0x000000F0 B: 0x0000000F

		A: 0x0000F000
DDPF_RGB	16	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	16	R: 0x0000001F G: 0x000007E0 B: 0x0000F800 A: 0x00000000
DDPF_RGB	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00008000
DDPF_RGB	24	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	24	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000

DDPF_RGB	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0xFF000000
DDPF_RGB | DDPF_ALPHAPIXELS	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0xFF000000

The HEL can create these formats in system memory. The DirectDraw device driver for a 3D-accelerated display card may create textures of other formats in display memory. Such a driver exports the **DDSCAPS_TEXTURE** flag to indicate that it can create textures.

Off-Screen Surface Formats

The following table shows the pixel formats for off-screen plain surfaces supported by the DirectX 3 HEL. The Masks column contains the red, green, blue, and alpha masks for each set of pixel format flags and bit depths.

Pixel format flags	Bit depth	Masks
DDPF_RGB | DDPF_PALETTEINDEXED1	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4	4	R: 0x00000000 G: 0x00000000

		B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED8	8	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB	16	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	24	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	24	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000

A: 0x00000000

In addition to supporting a wider range of off-screen surface formats, the HEL also supports surfaces intended for use by Direct3D, or other 3D renderers.

DirectDraw Interface Overviews

This section contains general information about the following DirectDraw COM interfaces:

- *IDirectDraw2 Interface*
- *IDirectDrawClipper Interface*
- *IDirectDrawPalette Interface*
- *IDirectDrawSurface2 Interface*

IDirectDraw2 Interface

The following topics contain additional information related to the *IDirectDraw2* interface:

- *DirectDraw Objects*
- *What's New in IDirectDraw2?*
- *Multiple DirectDraw Objects per Process*
- *Support for High Resolutions and True-Color Bit Depths*
- *Primary Surface Resource Sharing Model*
- *Changing Modes and Exclusive Access*
- *Creating DirectDraw Objects by Using CoCreateInstance*

DirectDraw Objects

DirectDraw objects represent the display hardware. An object is hardware-accelerated if the display device for which it was instantiated has hardware acceleration. A DirectDraw object can create three types of objects: DirectDrawSurface, DirectDrawPalette, and DirectDrawClipper.

More than one DirectDraw object can be instantiated at a time. The simplest example of this is using two monitors on a Windows 95 system. Although Windows 95 does not support dual monitors on its own, it is possible to write a DirectDraw HAL for each display device. The display device Windows 95 and GDI recognizes is the one that will be used when the default DirectDraw object is instantiated. The display device that Windows 95 and GDI do not recognize can be addressed by another, independent DirectDraw object that must be created by

using the second display device's identifying 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 IDirectDraw2?

The COM model that DirectX uses specifies that new functionality can be added by providing new interfaces. The *IDirectDraw2* interface supersedes the **IDirectDraw** interface. This new interface can be obtained by using the **IDirectDraw::QueryInterface** method, as shown in the following example:

```
// Create an IDirectDraw2 interface.
LPDIRECTDRAW  lpDD;
LPDIRECTDRAW2 lpDD2;

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

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

ddrval = lpDD->QueryInterface(IID_IDirectDraw2,
    (LPVOID *)&lpDD2);
if(ddrval != DD_OK)
    return;

ddscaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddrval = lpDD2->GetAvailableVidMem(&ddscaps, &total,
    &free);
if(ddrval != DD_OK)
    return;
```

This example shows C++ syntax for creating an **IDirectDraw** interface, which then uses the **IDirectDraw::QueryInterface** method to create an **IDirectDraw2** interface. This interface contains the **IDirectDraw2::GetAvailableVidMem** method. An attempt to use this method from an **IDirectDraw** interface will result in an error during compiling.

The **IDirectDraw2::GetAvailableVidMem** method is the only method that was added to the **IDirectDraw** interface when the **IDirectDraw2** interface was created. Two methods—**IDirectDraw2::SetDisplayMode** and

IDirectDraw2::EnumDisplayModes—were modified or extended when they were included in **IDirectDraw2**, however.

The interaction between the **IDirectDraw::SetCooperativeLevel** and **IDirectDraw::SetDisplayMode** methods is slightly different from the interaction between the **IDirectDraw2::SetCooperativeLevel** and **IDirectDraw2::SetDisplayMode** methods. If you use the **IDirectDraw** interface and an application gains exclusive (full-screen) mode by calling **IDirectDraw::SetCooperativeLevel** with the **DDSCL_EXCLUSIVE** flag, changes the mode by using **IDirectDraw::SetDisplayMode**, and then releases exclusive mode by calling **IDirectDraw::SetCooperativeLevel** with the **DDSCL_NORMAL** flag, the original display mode will not be restored. The new display mode will remain until the application calls the **IDirectDraw::RestoreDisplayMode** method or the DirectDraw object is deleted. However, if you are using the **IDirectDraw2** interface and an application follows the same steps, the original display mode will be restored when exclusive mode is lost.

Because some methods might change with the release of a new interface, mixing methods from an interface and its replacement (between **IDirectDraw** and **IDirectDraw2**, for example) can cause unpredictable results. You should use methods from only one version of an interface at a time.

Multiple DirectDraw Objects per Process

DirectDraw allows a process to call the **DirectDrawCreate** function as many times as necessary. A unique and independent interface 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.

Because the DirectDraw objects are independent, the DirectDrawSurface, DirectDrawPalette, and DirectDrawClipper objects created with a particular DirectDraw object should not be used with other DirectDraw objects because these objects are automatically released when the DirectDraw object is destroyed. If they are used with another DirectDraw object, they may stop functioning if the original object is destroyed.

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.

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 modes.

DirectDraw also supports HEL blitting of 24- and 32-bpp 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.

Windows allows a user to specify the type of monitor that is being used. 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 **IDirectDraw2::SetDisplayMode** method fails. Only modes that are supported on the installed monitor will be enumerated when you call the **IDirectDraw2::EnumDisplayModes** method.

Primary Surface Resource Sharing Model

DirectDraw has a simple resource sharing model. Display memory is a scarce, shared resource. If the display mode changes, all of the surfaces stored in display memory are lost. (For more information, see *Losing Surfaces*.)

DirectDraw implicitly creates a GDI primary surface when it is instantiated for a display device that DirectDraw is sharing with GDI. GDI is granted shared access to the primary surface. DirectDraw keeps track of the surface memory that GDI recognizes as the primary surface. The DirectDrawSurface object that owns GDI's primary surface can always be obtained by using the **IDirectDraw2::GetGDISurface** method.

GDI cannot cache fonts, brushes, and device-dependent bitmaps (DDBs) in the display memory managed by DirectDraw. The HAL must reserve whatever display memory the DIB engine driver needs before describing the available memory to DirectDraw's heap manager or before the display device driver can allocate and free memory for its cached data from DirectDraw's heap manager.

Changing Modes and Exclusive Access

An application can change display modes by using the **IDirectDraw2::SetDisplayMode** method. Modes can be changed by more than one application as long as they are all sharing a display card.

An application can change the pixel depth of the display mode only if it has obtained 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 **IDirectDrawSurface2::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.

Creating DirectDraw Objects by Using CoCreateInstance

You can create a DirectDraw object by using the **CoCreateInstance** function and the **IDirectDraw2::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 **IDirectDraw2::Initialize** method.

```
ddrval = CoCreateInstance(&CLSID_DirectDraw,
    NULL, CLSCTX_ALL, &IID_IDirectDraw2, &lpdd);
if(!FAILED(ddrval))
    ddrval = IDirectDraw2_Initialize(lpdd, NULL);
```

In this call to **CoCreateInstance**, the first parameter, *CLSID_DirectDraw*, is the class identifier of the DirectDraw driver object class, the *IID_IDirectDraw2* parameter identifies the particular DirectDraw interface to be created, and the *lpdd* parameter points to the DirectDraw object that is retrieved. If the call is successful, this functions returns an uninitialized object.

3 Before you use the DirectDraw object, you must call **IDirectDraw2::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 **IDirectDraw2::Initialize** method before using one of the methods associated with the DirectDraw object, a DDERR_NOTINITIALIZED error will occur.

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

```
CoUnitialize();
```

IDirectDrawClipper Interface

The *IDirectDrawClipper* interface simplifies the task of creating and maintaining a clip list. This interface is useful to applications running in a window, rather than in exclusive (full-screen) mode. Applications running in a window can use clip lists to limit screen updates to areas that have changed.

The following topics contain additional information related to the **IDirectDrawClipper** interface:

- *Clip Lists*
- *Sharing DirectDrawClipper Objects*
- *Driver-Independent DirectDrawClipper Objects*
- *Creating DirectDrawClipper Objects with CoCreateInstance*

Clip Lists

DirectDraw manages *clip lists* by using the DirectDrawClipper object. A clip list is a series of rectangles that describes the visible areas of the surface. A DirectDrawClipper object can be attached to any surface. A window handle can also be attached to a DirectDrawClipper object, in which case DirectDraw updates the DirectDrawClipper clip list with the clip list from the window as it changes.

Although the clip list is visible from the DirectDraw HAL, DirectDraw calls the HAL only for blitting with rectangles that meet the clip list requirements. For instance, if the upper-right rectangle of a surface was clipped and the application directed DirectDraw to blit the surface onto the primary surface, DirectDraw would have the HAL do two blits, the first being the upper-left corner of the surface, and the second being the bottom half of the surface.

The HAL considers the clip list for overlays only if the overlay hardware can support clipping and if destination color keying is not active. Most of today's hardware does not support occluded overlays unless they are subject to destination color keying. This can be reported to DirectDraw as a driver capability, in which case the overlay will be turned off if it becomes occluded. Under these conditions, the HAL does not consider clip lists either.

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 **IDirectDrawSurface2::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 **IDirectDrawSurface2::SetClipper** with a NULL clipper interface pointer, the reference count of the surface's DirectDrawClipper object will be decremented.

Note

If **IDirectDrawSurface2::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.

Driver-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 **IDirectDraw2::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 **IDirectDraw2::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, shut down COM by using the **CoUninitialize** function.

```
CoUnitialize();
```

IDirectDrawPalette Interface

The following topics contain additional information related to the *IDirectDrawPalette* interface:

- *DirectDrawPalette Objects*
- *Setting Palettes on Non-Primary Surfaces*
- *Sharing Palettes*
- *Palette Types*

DirectDrawPalette Objects

The DirectDrawPalette object is provided to enable direct manipulation of 16- and 256-color palettes. (A DirectDrawPalette object is typically attached to a DirectDrawSurface object.) A DirectDrawPalette object reserves entries 0 through 255 for 256-color palettes; it does not reserve any entries for 16-color palettes. It allows direct manipulation of the *color table* as a table. A color table is an array of color values (typically RGB triplets). This table can contain 16- or 24-bit RGB entries representing the colors associated with each of the indexes. For 16-color palettes, the table can also contain indexes to another 256-color palette.

An application can retrieve the entries in these tables by using the **IDirectDrawPalette::GetEntries** method, and it can change these entries by

using the **IDirectDrawPalette::SetEntries** method. This method has a *dwFlags* parameter that specifies when the changes to the palette should take effect.

You can choose between two methods for providing straightforward palette animation using DirectDrawPalette objects. Using the first method, you change the palette entries that correspond to the colors that need to be animated. You can do this with a single call to the **IDirectDrawPalette::SetEntries** method. The second method requires two DirectDrawPalette objects. The application performs the animation by attaching one object after the other to the DirectDrawSurface object. You can do this by using the **IDirectDrawSurface2::SetPalette** method.

Setting Palettes on Non-Primary 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. Furthermore, the **IDirectDrawSurface2::GetDC** method also ignores any DirectDrawPalette object selected into the surface.

Non-primary surface palettes are intended for use by applications or Direct3D (or other 3D renderers).

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 **IDirectDrawSurface2::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 **IDirectDrawSurface2::SetPalette** with a NULL palette interface pointer, the reference count of the surface's palette will be decremented.

If **IDirectDrawSurface2::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 Types

DirectDraw supports 1-bit (2 entry), 2-bit (4 entry), 4-bit (16 entry), and 8-bit (256 entry) palettes. A palette can be attached only to a surface with 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.

It is also possible to create *indexed palettes*. An indexed palette is one whose entries do not hold RGB colors, but rather integer indices into the array of **PALETTEENTRY** structures of some target palette. An indexed palette's color table is an array of 2, 4, 16, or 256 bytes, where each byte is an index into some unspecified destination palette.

To create an indexed palette, specify the **DDPCAPS_8BITENTRIES** flag when calling the **IDirectDraw2::CreatePalette** method. For example, to create a 4-bit indexed palette, specify **DDPCAPS_4BIT** |**DDPCAPS_8BITENTRIES**. When you create an indexed palette, 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 **IDirectDraw2::CreatePalette** method.

Using DirectDraw Palettes in Windowed Mode

The *IDirectDrawPalette* interface methods write directly to the hardware when the display is in exclusive (full-screen) mode. However, when the display is in nonexclusive (windowed) mode, the **IDirectDrawPalette** interface methods call the GDI's palette handling functions to work cooperatively with other windowed applications. This affects how you use palettes in a windowed DirectDraw application in the following ways:

- You must set the **peFlags** member of the **PALETTEENTRY** structure (passed to the **IDirectDraw2::CreatePalette** and **IDirectDrawPalette::SetEntries** methods) carefully.
- You should not attempt to alter the Windows static palette entries (indices 0 through 9 and 246 through 255).

The discussion in the following topics assumes that you have created a primary surface and a typical Windows window and that the Windows desktop is in an 8-bit palettized mode.

- Types of Palette Entries in Windowed Mode
- Calling IDirectDraw::CreatePalette in Windowed Mode
- Calling IDirectDrawPalette::SetEntries in Windowed Mode

Types of Palette Entries in Windowed Mode

In nonexclusive (windowed) mode, each type of palette entry must have a different set of **peFlags** members in the corresponding **PALETTEENTRY** structure in the array that you pass to the **IDirectDraw2::CreatePalette** or **IDirectDrawPalette::SetEntries** methods. In exclusive (full-screen) mode, you do not need to worry about the **peFlags** member; it is ignored. In nonexclusive (windowed) mode, the entries in your palette are one of the following three types:

- *Windows static entries.* Typically, Windows reserves entries 0 through 9 and 246 through 255 for itself and will not allow any application to change the color values for those entries. Although an application could build a 256-entry palette that includes these colors by calling the **GetSystemPaletteEntries** Win32 function, it is more effective for an application to refer to these entries directly. The application can do this by specifying which physical palette index a given entry in the palette table should map to, store the PC_EXPLICIT flag in the **peFlags** member of the **PALETTEENTRY** structure, and then set the **peRed** member equal to the index in the physical palette to which the entry should map. An application could also call the **SetSystemPaletteUse** Win32 function to force Windows to reserve only entries 0 and 255. In this case, you should set only entries 0 and 255 of your **PALETTEENTRY** structure to PC_EXPLICIT.
- *Animated entries.* These are entries your application can change to create palette animation effects. If the application specifies the PC_RESERVED flag for an animated entry, Windows will not allow any other application to map its logical palette entry to that physical entry. This prevents other applications from cycling their colors when your application sets a different color in that entry.
- *Non-animated entries.* These are entries your application will not change. Non-animated entries are simply filled with the PC_NOCOLLAPSE flag, which tells Windows not to substitute some other already-allocated physical palette entry.

In short, you should define the three types of entries in the **PALETTEENTRY** structure as follows:

Entry type	peFlags values	peRed, peGreen, and peBlue values
Windows static (indices 0-9 and 246-255 or 0 and 255)	PC_EXPLICIT	peRed = index, peGreen = 0, and peBlue = 0
Animated entries	PC_RESERVED | PC_NOCOLLAPSE	Color values
Non-animated entries	PC_NOCOLLAPSE	Color values

Calling IDirectDraw2::CreatePalette in Windowed Mode

The following example illustrates how to create a DirectDraw palette in nonexclusive (windowed) mode. It is vital that you set up every one of the 256 entries in the **PALETTEENTRY** structure that you submit to the **IDirectDraw2::CreatePalette** method.

```
LPDIRECTDRAW        lpDD; // Assumed to be initialized previously
PALETTEENTRY        pPaletteEntry[256];
int                 index;
HRESULT             ddrval;
LPDIRECTDRAWPALETTE lpDDPal;

// First set up the Windows static entries.
for (index = 0; index < 10 ; index++)
```

```
{
    // The first 10 static entries:
    pPaletteEntry[index].peFlags = PC_EXPLICIT;
    pPaletteEntry[index].peRed = index;
    pPaletteEntry[index].peGreen = 0;
    pPaletteEntry[index].peBlue = 0;

    // The last 10 static entries:
    pPaletteEntry[index+246].peFlags = PC_EXPLICIT;
    pPaletteEntry[index+246].peRed = index+246;
    pPaletteEntry[index+246].peGreen = 0;
    pPaletteEntry[index+246].peBlue = 0;
}

// Now set up private entries. In this example, the first 16
// available entries are animated.
for (index = 10; index < 26; index ++)
{
    pPaletteEntry[index].peFlags = PC_NOCOLLAPSE|PC_RESERVED;
    pPaletteEntry[index].peRed = 255;
    pPaletteEntry[index].peGreen = 64;
    pPaletteEntry[index].peBlue = 32;
}

// Now set up the rest, the non-animated entries.
for (; index < 246; index ++) // Index is set up by previous for loop
{
    pPaletteEntry[index].peFlags = PC_NOCOLLAPSE;
    pPaletteEntry[index].peRed = 25;
    pPaletteEntry[index].peGreen = 6;
    pPaletteEntry[index].peBlue = 63;
}

// All 256 entries are filled. Create the palette.
ddrval = lpDD->CreatePalette(DDPCAPS_8BIT, pPaletteEntry,
    &lpDDPal,NULL);
```

Calling IDirectDrawPalette::SetEntries in Windowed Mode

The rules that apply to the **PALETTEENTRY** structure used with the **IDirectDraw2::CreatePalette** method also apply to the **IDirectDrawPalette::SetEntries** method. Typically, you will maintain your own array of **PALETTEENTRY** structures, so you will not need to rebuild it. When necessary, you can modify the array, and then call **IDirectDrawPalette::SetEntries** when it is time to update the palette.

In most circumstances, do not attempt to set any of the Windows static entries when in nonexclusive (windowed) mode or you will get unpredictable results. The only exception is when you reset the all 256 entries. For palette animation, you typically change only a small subset of entries in your **PALETTEENTRY**

array. You submit only those entries to **IDirectDrawPalette::SetEntries**. If you are resetting such a small subset, you must reset only those entries marked with the PC_NOCOLLAPSE and PC_RESERVED flags. Attempting to animate other entries can have unpredictable results.

The following example illustrates palette animation in nonexclusive mode:

```
LPDIRECTDRAW        lpDD;        // Assumed to be initialized previously
PALETTEENTRY pPaletteEntry[256]; // Assumed to be initialized previously
LPDIRECTDRAWPALETTE lpDDPal;     // Assumed to be initialized previously
int                 index;
HRESULT             ddrval;
PALETTEENTRY        temp;

// Animate some entries. Cycle the first 16 available entries.
// They were already animated.
temp = pPaletteEntry[10];
for (index = 10; index < 25; index ++)
{
    pPaletteEntry[index] = pPaletteEntry[index+1];
}
pPaletteEntry[25] = temp;

// Set the values. Do not pass a pointer to the entire palette entry
// structure, but only to the changed entries.
ddrval = lpDDPal->SetEntries(
    0,                      // Flags must be zero
    10,                     // First entry
    16,                     // Number of entries
    & (pPaletteEntry[10])); // Where to get the data
```

IDirectDrawSurface2 Interface

The following topics contain additional information related to the *IDirectDrawSurface2* interface:

- *DirectDrawSurface Objects*
- *What's New in IDirectDrawSurface2?*
- *Creating Surfaces*
- *Frame-Buffer Access*
- *Flipping Surfaces and GDI's Frame Rate*
- *Losing Surfaces*
- *Color and Format Conversion*
- *Color Keying*
- *Overlay Z-Order*

- *Multiple Palettes for Off-Screen Surfaces*
- *Blitting to and from System Memory Surfaces*

DirectDrawSurface Objects

The DirectDrawSurface object represents a 2D piece of memory that contains data. This data is in a form understood by the display hardware represented by the DirectDraw object that created the DirectDrawSurface object. A DirectDrawSurface object is created by using the **IDirectDraw2::CreateSurface** method. The DirectDrawSurface object usually resides in the display memory of the display card, although this is not required. Unless specifically instructed otherwise during the creation of the DirectDrawSurface object, the 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.

DirectDrawSurface objects recognize, and are integrated with, the rest of the components of the Windows display system. DirectDrawSurface objects can create handles to Windows GDI device contexts (HDCs) that allow GDI functions to write to the surface memory represented by the DirectDrawSurface object. GDI perceives these HDCs as memory device contexts, but the hardware accelerators are usually enabled for them if they are in display memory.

What's New in IDirectDrawSurface2?

The COM model that DirectX uses specifies that new functionality can be added by providing new interfaces. The *IDirectDrawSurface2* interface supersedes the **IDirectDrawSurface** interface. This new interface can be obtained by using the **IDirectDraw::QueryInterface** method, as shown in the following example:

```
LPDIRECTDRAWSURFACE  lpSurf;
LPDIRECTDRAWSURFACE2 lpSurf2;

// Create surfaces.
memset(&ddsd, 0, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
    DDSCAPS_SYSTEMMEMORY;
ddsd.dwWidth = 10;
ddsd.dwHeight = 10;

ddrval = lpDD2->CreateSurface(&ddsd, &lpSurf,
    NULL);
```

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

ddrval = lpSurf->QueryInterface(
    IID_IDirectDrawSurface2, (LPVOID *)&lpSurf2);
if(ddrval != DD_OK)
    return;

ddrval = lpSurf2->PageLock(0);
if(ddrval != DD_OK)
    return;

ddrval = lpSurf2->PageUnlock(0);
if(ddrval != DD_OK)
    return;
```

The **IDirectDrawSurface2** interface contains all of the methods provided in the **IDirectDrawSurface** interface, as well as three new methods: **IDirectDrawSurface2::GetDDInterface**, **IDirectDrawSurface2::PageLock**, and **IDirectDrawSurface2::PageUnlock**.

Creating Surfaces

The DirectDrawSurface object represents a surface (pixel memory) that usually resides in the display memory, but the surface can exist in system memory if display memory is exhausted or if it is explicitly requested. If the hardware cannot support the capabilities requested or if it previously allocated those resources to another DirectDrawSurface object, a call to **IDirectDraw2::CreateSurface** will fail.

The **IDirectDraw2::CreateSurface** method usually creates one DirectDrawSurface object. If the DDSCAPS_FLIP flag in the **dwCaps** member of the **DDSCAPS** structure is set, the **IDirectDraw2::CreateSurface** method creates several DirectDrawSurface objects, referred to collectively as a *complex surface*. The additional surfaces created are also referred to as implicit surfaces. Implicit surfaces cannot be detached. For more information, see **IDirectDrawSurface2::DeleteAttachedSurface**.

DirectDraw does not permit the creation of display memory surfaces wider than the primary surface.

The following are examples of valid surface creation scenarios:

Scenario 1

The primary surface is the surface currently visible to the user. When you create a primary surface, you are actually creating a DirectDrawSurface object to access an already existing surface being used by GDI. Consequently, while all other types of surfaces require values for the **dwHeight** and **dwWidth** member of the

DDSURFACEDESC structure, a primary surface must not have them specified, even if you know they are the same dimensions as the existing surface.

The members of the **DDSURFACEDESC** structure (**ddsd** below) relevant to the creation of the primary surface are then filled.

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof(ddsd);

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

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

Scenario 2

Create a simple off-screen surface of the type that might be used to cache bitmaps that will later be composed with the blitter. A height and width are required for all surfaces except primary surfaces. The members of the **DDSURFACEDESC** structure (**ddsd** below) relevant to the creation of a simple off-screen surface are then filled.

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof(ddsd);

// 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.
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
dwHeight = 100;
dwWidth = 100;
```

DirectDraw creates this surface in display memory unless it will not fit, in which case the surface is created in system memory. If the surface must be created in one or the other, use the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY flags in **dwCaps** member of the **DDSCAPS** structure to specify system memory or display memory, respectively. An error is returned if the surface cannot be created in the specified location.

DirectDraw also allows for the creation of complex surfaces. A complex surface is a set of surfaces created with a single call to the **IDirectDraw2::CreateSurface** method. If the DDSCAPS_COMPLEX flag is set in the **IDirectDraw2::CreateSurface** call, one or more implicit surfaces will be created by DirectDraw in addition to the surface explicitly specified. Complex surfaces are managed as a single surface—a single call to the

IDirectDraw::Release method releases all surfaces in the structure, and a single call to the **IDirectDrawSurface2::Restore** method restores them.

Scenario 3

One of the most useful complex surfaces you can specify is composed of a primary surface and one or more back buffers that form a surface flipping environment. The members of the **DDSURFACEDESC** structure (**ddsd** below) relevant to complex surface creation are filled in to describe a flipping surface that has one back buffer.

```
DDSURFACEDESC ddsd;
ddsd.dwSize = sizeof(ddsd);

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

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

The previous statements construct a double-buffered flipping environment—a single call to the **IDirectDrawSurface2::Flip** method exchanges the surface memory of the primary surface and the back buffer. If you specify 2 for the value of the **BackBufferCount** member of the **DDSURFACEDESC** structure, two back buffers are created, and each call to **IDirectDrawSurface2::Flip** rotates the surfaces in a circular pattern, providing a triple-buffered flipping environment.

Frame-Buffer Access

DirectDrawSurface objects represent surface memory in the DirectDraw architecture. A DirectDrawSurface object allows an application to gain direct access to this surface memory by using the **IDirectDrawSurface2::Lock** method. An application calls this method, providing a **RECT** structure that specifies the rectangle on the surface it requires access to. If the application calls **IDirectDrawSurface2::Lock** with a NULL **RECT** structure, it is assumed that the application is requesting exclusive access to the entire piece of surface memory. This method fills in a **DDSURFACEDESC** structure with the information needed for the application to gain access to the surface memory. This information includes the pitch (or stride) and the pixel format of the surface, if different from the pixel format of the primary surface. When an application is finished with the surface memory, the surface memory can be made available again by using the **IDirectDrawSurface2::Unlock** method.

The following list describes some tips for avoiding the most common problems with rendering directly into a DirectDrawSurface object:

- Never assume a constant display pitch. Always examine the pitch information returned by the **IDirectDrawSurface2::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.
- Limit activity between the calls to the **IDirectDrawSurface2::Lock** and **IDirectDrawSurface2::Unlock** methods. The **IDirectDrawSurface2::Lock** method holds the WIN16 lock so that gaining access to surface memory can occur safely, and the **IDirectDrawSurface2::GetDC** method implicitly calls **IDirectDrawSurface2::Lock**. The WIN16 lock serializes access to GDI and USER, shutting down Windows for the duration between the **IDirectDrawSurface2::Lock** and **IDirectDrawSurface2::Unlock** operations, as well as between the **IDirectDrawSurface2::GetDC** and **IDirectDrawSurface2::ReleaseDC** operations.
- Copy aligned to display memory. Windows 95 uses 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 unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.

Flipping Surfaces and GDI's Frame Rate

Any surface in DirectDraw can be constructed as a *flipping surface*. A flipping surface is simply any piece of memory that can be swapped between a front buffer and a back buffer. Constructing a DirectDraw surface as a flipping surface has many advantages over the traditional, limited scope of page flipping.

When an application uses the **IDirectDrawSurface2::Flip** method to request a flip operation, the surface memory areas associated with the DirectDrawSurface objects being flipped are switched. Surfaces attached to the DirectDrawSurface objects being flipped are not affected. For example, in a double-buffered situation, an application that draws on the back buffer always uses the same DirectDrawSurface object. The surface memory underneath the object is simply switched with the front buffer when **IDirectDrawSurface2::Flip** is requested.

If the front buffer is visible, either because it is the primary surface or because it is an overlay that is currently visible, subsequent calls to the **IDirectDrawSurface2::Lock** or **IDirectDrawSurface2::Blt** methods that target the back buffer fail with the **DDERR_WASSTILLDRAWING** return value until the next vertical refresh occurs. This occurs because the front buffer's previous surface memory, which is no longer attached to the back buffer, is still being drawn to the physical display by the hardware. This situation disappears during the next vertical refresh because the hardware that updates the physical display re-reads the location of the display memory on every refresh.

This physical requirement makes calling the **IDirectDrawSurface2::Flip** method on visible surfaces an asynchronous command. When building games, you should, for example, perform all of the non-visual elements of the game after this

method is called. When the input, audio, game-play, and system-memory drawing operations have been completed, you can begin the drawing tasks that require gaining access to the visible back buffers.

When your application needs to run in a window and still requires a flipping environment, it will attempt to create a flipping overlay surface. If the hardware does not support overlays, you can create a primary surface that page flips. When a surface is about to become the primary surface and GDI does not have information about that surface, you can blit the contents of the primary surface that GDI is writing to onto the buffer that is about to become visible. This takes little, if any, processing time because the blits are performed asynchronously. It can, however, consume considerable blitter bandwidth that is dependent on screen resolution and the size of the window that is being page flipped. As long as the frame rate does not dip below 20 frames per second, GDI will appear to be operating correctly.

Before you instantiate a DirectDraw object, GDI is already using your display memory to display itself. When you call DirectDraw to instantiate a primary surface, the memory address of that surface will be the same as GDI is currently using.

If you create a complex surface with a back buffer, GDI will first point to the display memory for the primary surface. Because GDI was created before DirectDraw, GDI cannot be informed of DirectDraw's actions. Therefore, GDI will continue operating on this surface, even if you have flipped it and it is now the non-visible back buffer.

Many applications begin by creating one large window that covers the entire screen. As long as your application is active and has the focus, GDI will not attempt to write into its copy of the buffer because nothing it controls needs redrawing.

For other scenarios, remember that GDI has information about the original surface only, and it has no information about whether it is currently the primary surface or a back buffer. If you do not need the GDI screen, you can use the technique described above. If you do need GDI, you can try the following technique:

1. Create a primary surface with two back buffers.
2. Blit the initial primary surface (the GDI surface) to the middle back buffer.
3. Flip the surfaces (with the *lpDDSurfaceTargetOverride* parameter set to NULL) to put GDI into last place and make your initial copy visible.

After you have done this, you can copy from the GDI buffer to the middle buffer, draw what you want the user to see on that buffer, you can then keep GDI safely on the bottom and oscillate between the other two buffers, by using the following example:

```
pPrimary->Flip(pMiddle);
```

Losing 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 of the methods return **DDERR_SURFACELOST** and perform no other action.

Surfaces can be lost because the display card mode was changed or because an application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. The **IDirectDrawSurface2::Restore** method re-creates these lost surfaces and reconnects them to their DirectDrawSurface objects.

For more information, see *Changing Modes and Exclusive Access*.

Color and Format Conversion

Non-RGB surface formats are described by four-character codes (FOURCC codes). If an application calls the **IDirectDrawSurface2::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**, **dwYBits**, **dwUBits**, **dwVBits**, and **dwYUVAlphaBits** 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 **dwRGBBitCount**, **dwRBits**, **dwGBits**, **dwBBits**, and **dwRGBAlphaBits** 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 non-standard 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 what the blitting hardware is capable of; the other represents what the overlay hardware is capable of.

Color Keying

Source and destination color keying for blits and overlays are supported by DirectDraw. You can supply a color key or a color range for both of these types of color keying.

Source color keying specifies a color or color range that, in the case of blitting, is not copied, or, in the case of overlays, is not visible on the destination. *Destination color keying* specifies a color or color range that, in the case of

blitting, is replaced or, in the case of overlays, is covered up on the destination. The source color key specifies what can and cannot be read from the surface. The destination color key specifies what can and cannot be written onto, or covered up, on the destination surface. If a destination surface has a color key, only the pixels that match the color key are be changed, or covered up, on the destination surface.

Some hardware supports color ranges only for YUV pixel data. YUV data is usually video, and the transparent background may not be a single color due to quantitization errors during conversion. Content should be written to a single transparent color whenever possible, regardless of pixel format.

Color keys are specified in the pixel format of a surface. If a surface is in a palettized format, the color key is specified 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. The *dwFlags* parameter of the **IDirectDrawSurface2::SetColorKey** method specifies whether the color key is to be used for overlay or blit operations, and whether it is a source or a destination key. 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);
```

Overlay Z-Order

Overlay z-order determines the order in which overlays clip each other, enabling a hardware sprite system to be implemented under DirectDraw. Overlays are assumed to be on top of all other screen components. 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 it. Overlays that do not have a specified z-order behave in unpredictable ways when overlaying the same area on the primary surface. Finally, overlays without a specified z-order are assumed to have a z-order of 0. The possible z-order of overlays ranges from 0, which is just on top of the primary surface, to 4 billion, which is as close to the viewer as possible. An overlay with a z-order of 2 would obscure an overlay with a z-order of 1. No two overlays can have the same z-order.

Multiple Palettes for Off-Screen Surfaces

DirectDraw allows you to create multiple palettes that can be attached to off-screen surfaces. When this is done, the off-screen surfaces no longer share the palette of the primary surface. If you create an off-screen surface with a pixel format that is different from the primary surface's, it is assumed that the hardware can use it. For example, if a palettized off-screen surface is created when the primary surface is in 16-bit color mode, it is assumed that the blitter can convert palettized surfaces to true color during the blit operation.

DirectDraw allows you to create standard 8-bit palettized surfaces, which can display 256 colors, and two kinds of 4-bit palettized surfaces, each of which can display 16 colors. The first type of 4-bit palettized surface is indexed into a true-color color table; the second type is indexed into the indexed color table for the primary surface. This second type of palette provides 50 percent compression and a layer of indirection to the sprites stored using it.

If these surfaces are to be created, the blitter must be able replace the palette during the blit operation. Therefore, when a blit operation occurs from one palettized surface to another, the palette is ignored. Palette decoding is done only to true-color surfaces, or when the 4-bit palette is an index to an index in the 8-bit palette. In all other cases, the indexed palette is the palette of the destination.

Raster operations for palettized surfaces are ignored. Changing the attached palette of a surface is a very quick operation. All three of these palettized surfaces should be supported as textures on 3D-accelerated hardware.

For information about the pixel formats for off-screen plain surfaces, see *Off-Screen Surface Formats*.

Blitting to and from System Memory Surfaces

Some display cards have DMA hardware that allows them to efficiently blit to and from system-memory surfaces. Drivers report this capability through the **DDCAPS** structure. This structure contains the following 12 new members:

dwSVBCaps	**dwVSBCaps**	**dwSSBCaps**
dwSVBCKeyCaps	**dwVSBCKeyCaps**	**dwSSBCKeyCaps**
dwSVBFXCaps	**dwVSBFXCaps**	**dwSSBFXCaps**

dwSVBRops **dwVSBRops** **dwSSBRops**

The SVB acronym indicates capability values that relate system-memory to display-memory blits, VSB for capability values that relate display-memory to system-memory blits, and SSB for capability values that relate to system-memory to system-memory blits.

The **dwSVBCaps** member corresponds to the **dwCaps** member except that it describes the blitting capabilities of the display driver for system-memory to display-memory blits. Similarly, the **dwSVBCKeyCaps** member corresponds to the **dwCKeyCaps** member, and the **dwSVBFXCaps** member corresponds to the **dwFXCaps** member. The **dwSVBRops** member array describes the raster operations the driver supports for this type of blit.

These members are valid only if the **DDCAPS_CANBLTSYSMEM** flag is set in **dwCaps**, indicating that the driver is able to blit to or from system memory.

If the system memory surface being used by the hardware blitter is not locked, DirectDraw automatically calls the **IDirectDrawSurface2::PageLock** method on the surface to ensure that the memory has been locked.

DirectDraw Tutorials

This section contains a series of tutorials, each of which provides step-by-step instructions for implementing a simple DirectDraw application. These tutorials use many of the DirectDraw sample files that are provided with this SDK. The sample files begin with a DDEX prefix. These samples demonstrate how to set up DirectDraw, and how to use the DirectDraw methods to perform simple tasks.

- *Tutorial 1: The Basics of DirectDraw* (DDEX1)
- *Tutorial 2: Loading Bitmaps on the Back Buffer* (DDEX2)
- *Tutorial 3: Blitting from an Off-Screen Surface* (DDEX3)
- *Tutorial 4: Color Keys and Bitmap Animation* (DDEX4)
- *Tutorial 5: Dynamically Modifying Palettes* (DDEX5)

The samples in these tutorials use the older **IDirectDraw** and **IDirectDrawSurface** interfaces. If you want to update these examples so they use the DirectX 3 interfaces—*IDirectDraw2* and *IDirectDrawSurface2*—add a **QueryInterface** for both interfaces, as described in *What's New in IDirectDraw2?* and *What's New in IDirectDrawSurface2?*. In addition, you must change the appropriate parameters of any methods that have been updated for **IDirectDraw2** or **IDirectDrawSurface2**.

Note

The DDEX samples files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the vtable and **this** pointers to the interface methods. For more information, see *Accessing COM Objects by Using C*.

Tutorial 1: The Basics of DirectDraw

To use DirectDraw, you first create an instance of the DirectDraw object, which represents the display adapter on the computer. You then use the interface methods to manipulate the object. In addition, you need to create one or more instances of a DirectDrawSurface object to be able to display your game on a graphics surface.

To demonstrate this, the DDEX1 sample included with this SDK performs the following steps:

- *Step 1: Creating a DirectDraw Object*
- *Step 2: Determining the Application's Behavior*
- *Step 3: Changing the Display Mode*
- *Step 4: Creating Flipping Surfaces*
- *Step 5: Rendering to the Surfaces*
- *Step 6: Writing to the Surface*
- *Step 7: Flipping the Surfaces*
- *Step 8: Deallocating the DirectDraw Objects*

Step 1: Creating a DirectDraw Object

To create an instance of a DirectDraw object, your application should use the **DirectDrawCreate** function as shown in the **doInit** function of the DDEX1 program. **DirectDrawCreate** contains three parameters. The first parameter takes a globally unique identifier (GUID) that represents the display device. The GUID, in most cases, is set to NULL, which means DirectDraw uses the default display driver for the system. The second parameter contains the address of a pointer that identifies the location of the DirectDraw object if it is created. The third parameter is always set to NULL and is included for future expansion.

The following example shows how to create the DirectDraw object and how to determine if the creation was successful or not:

```
ddrval = DirectDrawCreate(NULL, &lpDD, NULL);
if(ddrval == DD_OK)
{
    // lpDD is a valid DirectDraw object.
```

```
}
else
{
    // The DirectDraw object could not be created.
}
```

Step 2: Determining the Application's Behavior

Before you can change the resolution of your display, you must at a minimum specify the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags in the *dwFlags* parameter of the **IDirectDraw::SetCooperativeLevel** method. This gives your application complete control over the display device, and no other application will be able to share it. In addition, the DDSCL_FULLSCREEN flag sets the application in exclusive (full-screen) mode. Your application covers the entire desktop, and only your application can write to the screen. The desktop is still available, however. (For an example of how to see the desktop in an application running in exclusive mode, start DDEX1 and press ALT + TAB.)

The following example demonstrates the use of **IDirectDraw::SetCooperativeLevel**:

```
HRESULT      ddrval;
LPDIRECTDRAW lpDD;     // Already created by DirectDrawCreate

ddrval = lpDD->SetCooperativeLevel(hwnd, DDSCL_EXCLUSIVE |
    DDSCL_FULLSCREEN);
if(ddrval == DD_OK)
{
    // Exclusive mode was successful.
}
else
{
    // Exclusive mode was not successful.
    // The application can still run, however.
}
```

If **IDirectDraw::SetCooperativeLevel** does not return DD_OK, you can still run your application. The application will not be in exclusive mode, however, and it might not be capable of the performance your application requires. In this case, you might want to display a message that allows the user to decide whether or not to continue.

One requirement for using **IDirectDraw::SetCooperativeLevel** is that you must pass a handle to a window (**HWND**) to allow Windows to determine if your application terminates abnormally. For example, if a general protection (GP) fault occurs and GDI is flipped to the back buffer, the user will not be able to get the Windows screen back. To prevent this from occurring, DirectDraw provides a process running in the background that traps messages that are sent to that window. DirectDraw uses these messages to determine when the application

terminates. This feature imposes some restrictions, however. First, you have to specify the window handle that is retrieving messages for your application—that is, if you create another window, you must ensure that you specify the window that is active. Otherwise, you might experience problems, including unpredictable behavior from GDI, or no response when you press ALT+TAB.

Step 3: Changing the Display Mode

After you have set the application's behavior, you can use the **IDirectDraw::SetDisplayMode** method to change the resolution of the display. The following example shows how to set the display mode to 640×480×8 bpp:

```
HRESULT      ddrval;
LPDIRECTDRAW lpDD;  // Already created

ddrval = lpDD->SetDisplayMode(640, 480, 8);
if(ddrval == DD_OK)
{
    // The display mode changed successfully.
}
else
{
    // The display mode cannot be changed.
    // The mode is either not supported or
    // another application has exclusive mode.
}
```

When you set the display mode, you should ensure that if the user's hardware cannot support higher resolutions, your application reverts to a standard mode that is supported by a majority of display adapters. For example, your application could be designed to run on all systems that support 640×480×8 as a standard backup resolution. (**IDirectDraw::SetDisplayMode** returns a DDERR_INVALIDMODE error value if the display adapter could not be set to the desired resolution. Therefore, you should use the **IDirectDraw::EnumDisplayModes** method to determine the capabilities of the user's display adapter before trying to set the display mode.)

Step 4: Creating Flipping Surfaces

After you have set the display mode, you must create the surfaces on which to place your application. Because the DDEX1 example is using the **IDirectDraw::SetCooperativeLevel** method to set the mode to exclusive (full-screen) mode, you can create surfaces that flip between the surfaces. If you were using **IDirectDraw::SetCooperativeLevel** to set the mode to DDSCL_NORMAL, you could create only surfaces that blit between the surfaces. Creating flipping surfaces requires the following steps:

- Defining the surface requirements

- Creating the surfaces

Defining the Surface Requirements

The first step in creating flipping surfaces is to define the surface requirements in a **DDSURFACEDESC** structure. The following example shows the structure definitions and flags needed to create a flipping surface.

```
// Create the primary surface with one back buffer.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
    DDSCAPS_FLIP | DDSCAPS_COMPLEX;

ddsd.dwBackBufferCount = 1;
```

In this example, the **dwSize** member is set to the size of the **DDSURFACEDESC** structure. This is to prevent any DirectDraw method call you use from returning with an invalid member error. (The **dwSize** member was provided for future expansion of the **DDSURFACEDESC** structure.)

The **dwFlags** member determines which members in the **DDSURFACEDESC** structure will be filled with valid information. For the DDEX1 example, **dwFlags** is set to specify that you want to use the **DDSCAPS** structure (DDSD_CAPS) and that you want to create a back buffer (DDSD_BACKBUFFERCOUNT).

The **dwCaps** member in the example indicates the flags that will be used in the **DDSCAPS** structure. In this case, it specifies a primary surface (DDSCAPS_PRIMARYSURFACE), a flipping surface (DDSCAPS_FLIP), and a complex surface (DDSCAPS_COMPLEX).

Finally, the example specifies one back buffer. The back buffer is where the backgrounds and sprites will actually be written. The back buffer is then flipped to the primary surface. In the DDEX1 example, the number of back buffers is set to 1. You can, however, create as many back buffers as the amount of display memory allows. For more information on creating more than one back buffer, see *Triple Buffering*.

Surface memory can be either display memory or system memory. DirectDraw uses system memory if the application runs out of display memory (for example, if you specify more than one back buffer on a display adapter with only 1 MB of RAM). You can also specify whether to use only system memory or only display memory by setting the **dwCaps** member in the **DDSCAPS** structure to DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY. (If you specify DDSCAPS_VIDEOMEMORY, but not enough memory is available to create the surface, **IDirectDraw::CreateSurface** returns with a DDERR_OUTOFVIDEOMEMORY error.)

Creating the Surfaces

After the **DDSURFACEDESC** structure is filled, you can use it and *lpDD*, the pointer to the DirectDraw object that was created by the **DirectDrawCreate** function, to call the **IDirectDraw::CreateSurface** method, as shown in the following example:

```
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
if(ddrval == DD_OK)
{
    // lpDDSPrimary points to the new surface.
}
else
{
    // The surface was not created.s
    return FALSE;
}
```

The *lpDDSPrimary* parameter will point to the primary surface returned by **IDirectDraw::CreateSurface** if the call succeeds.

After the pointer to the primary surface is available, you can use the **IDirectDrawSurface::GetAttachedSurface** method to retrieve a pointer to the back buffer, as shown in the following example:

```
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddrval = lpDDSPrimary->GetAttachedSurface(&ddcaps, &lpDDSBack);
if(ddrval == DD_OK)
{
    // lpDDSBack points to the back buffer.
}
else
{
    return FALSE;
}
```

By supplying the address of the surface's primary surface and by setting the capabilities value with the DDSCAPS_BACKBUFFER flag, the *lpDDSBack* parameter will point to the back buffer if the **IDirectDrawSurface::GetAttachedSurface** call succeeds.

Step 5: Rendering to the Surfaces

After the primary surface and a back buffer have been created, the DDEX1 example renders some text on the primary surface and back buffer surface by using standard Windows GDI functions, as shown in the following example:

```
if (lpDDSPrimary->GetDC(&hdc) == DD_OK)
{
    SetBkColor(hdc, RGB(0, 0, 255));
```

```
    SetTextColor(hdc, RGB(255, 255, 0));
    TextOut(hdc, 0, 0, szFrontMsg, lstrlen(szFrontMsg));
    lpDDSPrimary->ReleaseDC(hdc);
}

if (lpDDSBack->GetDC(&hdc) == DD_OK)
{
    SetBkColor(hdc, RGB(0, 0, 255));
    SetTextColor(hdc, RGB(255, 255, 0));
    TextOut(hdc, 0, 0, szBackMsg, lstrlen(szBackMsg));
    lpDDSBack->ReleaseDC(hdc);
}
```

The example uses the **IDirectDrawSurface::GetDC** method to retrieve the handle to the device context, and it internally locks the surface. If you are not going to use Windows functions that require a handle to a device context, you could use the **IDirectDrawSurface::Lock** and **IDirectDrawSurface::Unlock** methods to lock and unlock the back buffer.

Locking the surface memory (whether the whole surface or part of a surface) ensures that your application and the system blitter cannot obtain access to the surface memory at the same time. This prevents errors from occurring while your application is writing to surface memory. In addition, your application cannot page flip until the surface memory is unlocked.

After the surface is locked, the example uses the standard **SetBkColor** Windows GDI function to set the background color, **SetTextColor** to select the color of the text to be placed on the background, and **TextOut** to print the text and background color on the surfaces.

After the text has been written to the buffer, the example uses the **IDirectDrawSurface::ReleaseDC** method to unlock the surface and release the handle. Whenever your application finishes writing to the back buffer, you must call either **IDirectDrawSurface::ReleaseDC** or **IDirectDrawSurface::Unlock**, depending on your application. Your application cannot flip the surface until the surface is unlocked.

Typically, you write to a back buffer, which you then flip to the primary surface to be displayed. In the case of DDEX1, there is a significant delay before the first flip, so DDEX1 writes to the primary buffer in the initialization function to prevent a delay before displaying the surface. As you will see in a subsequent step of this tutorial, the DDEX1 example writes only to the back buffer during WM_TIMER. An initialization function or title page may be the only place where you might want to write to the primary surface.

Note

After the surface is unlocked by using **IDirectDrawSurface::Unlock**, the pointer to the surface memory is invalid. You must use **IDirectDrawSurface::Lock** again to obtain a valid pointer to the surface memory.

Step 6: Writing to the Surface

The first half of the WM_TIMER message in DDEX1 is devoted to writing to the back buffer, as shown in the following example:

```
case WM_TIMER:
    // Flip surfaces.
    if(bActive)
    {
        if (lpDDSBack->GetDC(&hdc) == DD_OK)
        {
            SetBkColor(hdc, RGB(0, 0, 255));
            SetTextColor(hdc, RGB(255, 255, 0));
            if(phase)
            {
                TextOut(hdc, 0, 0, szFrontMsg, lstrlen(szFrontMsg));
                phase = 0;
            }
            else
            {
                TextOut(hdc, 0, 0, szBackMsg, lstrlen(szBackMsg));
                phase = 1;
            }
            lpDDSBack->ReleaseDC(hdc);
        }
```

The line of code that calls the **IDirectDrawSurface2::GetDC** method locks the back buffer in preparation for writing. The **SetBkColor** and **SetTextColor** functions set the colors of the background and text.

Next, the "phase" variable determines whether the primary buffer message or the back buffer message should be written. If "phase" equals 1, the primary surface message is written, and "phase" is set to 0. If "phase" equals 0, the back buffer message is written, and "phase" is set to 1. Note, however, that in both cases the messages are written to the back buffer.

After the message is written to the back buffer, the back buffer is unlocked by using the **IDirectDrawSurface::ReleaseDC** method.

Step 7: Flipping the Surfaces

After the surface memory is unlocked, you can use the **IDirectDrawSurface::Flip** method to flip the back buffer to the primary surface, as shown in the following example:

```
while(1)
{
    HRESULT ddrval;
    ddrval = lpDDSPrimary->Flip(NULL, 0);
    if(ddrval == DD_OK)
    {
        break;
    }
    if(ddrval == DDERR_SURFACELOST)
    {
        ddrval = lpDDSPrimary->Restore();
        if(ddrval != DD_OK)
        {
            break;
        }
    }
    if(ddrval != DDERR_WASSTILLDRAWING)
    {
        break;
    }
}
```

In the example, **lpDDSPrimary** designates the primary surface and its associated back buffer. When **IDirectDrawSurface::Flip** is called, the front and back surfaces are exchanged (only the pointers to the surfaces are changed; no data is actually moved). If the flip is successful and returns DD_OK, the application breaks from the while loop.

If the flip returns with a DDERR_SURFACELOST value, an attempt is made to restore the surface by using the **IDirectDrawSurface::Restore** method. If the restore is successful, the application loops back to the **IDirectDrawSurface::Flip** call and tries again. If the restore is unsuccessful, the application breaks from the while loop, and returns with an error.

When you call **IDirectDrawSurface::Flip**, the flip does not complete immediately. Rather, a flip is scheduled for the next time a vertical blank occurs on the system. If, for example, the previous flip has not occurred, **IDirectDrawSurface::Flip** returns DDERR_WASSTILLDRAWING. In the example, the **IDirectDrawSurface::Flip** call continues to loop until it returns DD_OK.

Step 8: Deallocating the DirectDraw Objects

When you press the F12 key, the DDEX1 application processes the WM_DESTROY message before exiting the application. This message calls the **finiObjects** function, which contains all of the **IUnknown::Release** calls, as shown below:

```
static void finiObjects(void)
```

```
{
    if(lpDD != NULL)
    {
        if(lpDDSPrimary != NULL)
        {
            lpDDSPrimary->Release();
            lpDDSPrimary = NULL;
        }
        lpDD->Release();
        lpDD = NULL;
    }
} // finiObjects
```

The application checks if the pointers to the DirectDraw object (**lpDD**) and the DirectDrawSurface object (**lpDDSPrimary**) are not equal to NULL. Then DDEX1 calls the **IDirectDrawSurface::Release** method to decrease the reference count of the DirectDrawSurface object by 1. Because this brings the reference count to 0, the DirectDrawSurface object is deallocated. The DirectDrawSurface pointer is then destroyed by setting its value to NULL. Next, the application calls **IDirectDraw::Release** to decrease the reference count of the DirectDraw object to 0, deallocating the DirectDraw object. This pointer is then also destroyed by setting its value to NULL.

Tutorial 2: Loading Bitmaps on the Back Buffer

The sample discussed in this tutorial (DDEX2) expands on the DDEX1 sample that was discussed in Tutorial 1. DDEX2 includes functionality to load a bitmap file on the back buffer. This new functionality is demonstrated in the following steps:

- *Step 1: Creating the Palette*
- *Step 2: Setting the Palette*
- *Step 3: Loading a Bitmap on the Back Buffer*
- *Step 4: Flipping the Surfaces*

As in DDEX1, **doInit** is the initialization function for the DDEX2 application. Although the code for the DirectDraw initialization does not look quite the same in DDEX2 as it did in DDEX1, it is essentially the same, except for the following section:

```
lpDDPal = DDLoadPalette(lpDD, szBackground);

if (lpDDPal == NULL)
    goto error;
```

```
ddrval = lpDDSPrimary->SetPalette(lpDDPal);

if(ddrval != DD_OK)
    goto error;

// Load a bitmap into the back buffer.
ddrval = DDReLoadBitmap(lpDDSBack, szBackground);

if(ddrval != DD_OK)
    goto error;
```

Step 1: Creating the Palette

The DDEX2 sample first loads the palette into a structure by using the following code:

```
lpDDPal = DDLoadPalette(lpDD, szBackground);

if (lpDDPal == NULL)
    goto error;
```

DDLoadPalette is part of the common DirectDraw functions found in the Ddutil.cpp file located in the \Dxsdk\Sdk\Samples\Misc directory. Most of the DirectDraw sample files in this SDK use this file. Essentially, it contains the functions for loading bitmaps and palettes from either files or resources. To avoid having to repeat code in the example files, these functions were placed in a file that could be reused. Make sure you include Ddutil.cpp in the list of files to be compiled with the rest of the DDEX samples.

For DDEX2, the **DDLoadPalette** function creates a DirectDrawPalette object from the Back.bmp file. The **DDLoadPalette** function determines if a file or resource for creating a palette exists. If one does not, it creates a default palette. For DDEX2, it extracts the palette information from the bitmap file and stores it in a structure pointed to by *ape*.

DDEX2 then creates the DirectDrawPalette object, as shown in the following example:

```
pdd->CreatePalette(DDPCAPS_8BIT, ape, &ddpal, NULL);
return ddpal;
```

When the **IDirectDraw::CreatePalette** method returns, the *ddpal* parameter points to the DirectDrawPalette object, which is then returned from the **DDLoadPalette** call.

The *ape* parameter is a pointer to a structure that can contain either 2, 4, 16, or 256 entries, organized linearly. The number of entries depends on the *dwFlags* parameter in the **IDirectDraw::CreatePalette** method. In this case, the *dwFlags* parameter is set to DDPCAPS_8BIT, which indicates that there are 256 entries in

this structure. Each entry contains 4 bytes (a red channel, a green channel, a blue channel, and a flags byte).

Step 2: Setting the Palette

After you create the palette, you pass the pointer to the DirectDrawPalette object (*ddpal*) to the primary surface by calling the **IDirectDrawSurface::SetPalette** method, as shown in the following example:

```
ddrval = lpDDSPrimary->SetPalette(lpDDPal);

if(ddrval != DD_OK)
    // SetPalette failed.
```

After you have called **IDirectDrawSurface::SetPalette**, the DirectDrawPalette object is associated with the DirectDrawSurface object. Any time you need to change the palette, you simply create a new palette and set the palette again. (Although this is how this is done in this example, there are other ways of changing the palette, as will be shown in later examples.)

Step 3: Loading a Bitmap on the Back Buffer

After the DirectDrawPalette object is associated with the DirectDrawSurface object, DDEX2 loads the Back.bmp bitmap on the back buffer by using the following code:

```
// Load a bitmap into the back buffer.
ddrval = DDReLoadBitmap(lpDDSBack, szBackground);

if(ddrval != DD_OK)
    // Load failed.
```

DDReLoadBitmap is another function found in Ddutil.cpp. It loads a bitmap from a file or resource into an already existing DirectDraw surface. (You could also use **DDLoadBitmap** to create a surface and load the bitmap into that surface. For more information, see *Tutorial 5: Dynamically Modifying Palettes*.) For DDEX2, it loads the Back.bmp file pointed to by *szBackground* onto the back buffer pointed to by *lpDDSBack*. The **DDReLoadBitmap** function calls the **DDCopyBitmap** function to copy the file onto the back buffer and stretch it to the proper size.

The **DDCopyBitmap** function copies the bitmap into memory, and it uses the **GetObject** function to retrieve the size of the bitmap. It then uses the following code to retrieve the size of the back buffer onto which it will place the bitmap:

```
// Get the size of the surface.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_HEIGHT | DDSD_WIDTH;
pdds->GetSurfaceDesc(&ddsd);
```

The *ddsd* value is a pointer to the **DDSURFACEDESC** structure. This structure stores the current description of the DirectDraw surface. In this case, the **DDSURFACEDESC** members describe the height and width of the surface, which are indicated by DDSD_HEIGHT and DDSD_WIDTH. The call to the **IDirectDrawSurface::GetSurfaceDesc** method then loads the structure with the proper values. For DDEX2, the values will be 480 for the height and 640 for the width.

The **DDCopyBitmap** function locks the surface and copies the bitmap to the back buffer, stretching or compressing it as applicable by using the **StretchBlt** function, as shown below:

```
if ((hr = pdds->GetDC(&hdc)) == DD_OK)
{
    StretchBlt(hdc, 0, 0, ddsd.dwWidth, ddsd.dwHeight, hdcImage, x, y,
        dx, dy, SRCCOPY);
    pdds->ReleaseDC(hdc);
}
```

Step 4: Flipping the Surfaces

Flipping surfaces in the DDEX2 sample is essentially the same process as that in the DDEX1 tutorial (see *Tutorial 1: The Basics of DirectDraw*) except that if the surface is lost (DDERR_SURFACELOST), the bitmap must be reloaded on the back buffer by using the **DDReLoadBitmap** function after the surface is restored.

Tutorial 3: Blitting from an Off-Screen Surface

The sample in Tutorial 2 (DDEX2) takes a bitmap and puts it in the back buffer, and then it flips between the back buffer and the primary buffer. This is not a very realistic approach to displaying bitmaps. The sample in this tutorial (DDEX3) expands on the capabilities of DDEX2 by including two off-screen buffers in which the two bitmaps—one for the even screen and one for the odd screen—are stored. It uses the **IDirectDrawSurface::BltFast** method to copy the contents of an off-screen surface to the back buffer, and then it flips the buffers and copies the next off-screen surface to the back buffer.

The new functionality demonstrated in DDEX3 is shown in the following steps:

- *Step 1: Creating the Off-Screen Surfaces*
- *Step 2: Loading the Bitmaps to the Off-Screen Surfaces*
- *Step 3: Blitting the Off-Screen Surfaces to the Back Buffer*

Step 1: Creating the Off-Screen Surfaces

The following code is added to the **doInit** function in DDEX3 to create the two off-screen buffers:

```
// Create an offscreen bitmap.
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 480;
ddsd.dwWidth = 640;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSOne, NULL);
if(ddrval != DD_OK)
{
    return initFail(hwnd);
}

// Create another offscreen bitmap.
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSTwo, NULL);
if(ddrval != DD_OK)
{
    return initFail(hwnd);
}
```

The **dwFlags** member specifies that the application will use the **DDSCAPS** structure, and it will set the height and width of the buffer. The surface will be an off-screen plain buffer, as indicated by the DDSCAPS_OFFSCREEN flag set in the **DDSCAPS** structure. The height and the width are set as 480 and 640, respectively, in the **DDSURFACEDESC** structure. The surface is then created by using the **IDirectDraw::CreateSurface** method.

Because both of the off-screen plain buffers are the same size, the only requirement for creating the second buffer is to call **IDirectDraw::CreateSurface** again with a different pointer name.

You can also specifically request that the off-screen buffer be placed in system memory or display memory by setting either the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY capability in the **DDSCAPS** structure. By saving the bitmaps in display memory, you can increase the speed of the transfers between the off-screen surfaces and the back buffer. This will become more important when using bitmap animation. However, if you specify DDSCAPS_VIDEOMEMORY for the off-screen buffer and not enough display memory is available to hold the entire bitmap, a DDERR_OUTOFVIDEOMEMORY error value will be returned when you attempt to create the surface.

Step 2: Loading the Bitmaps to the Off-Screen Surfaces

After the two off-screen surfaces are created, DDEX3 uses the **InitSurfaces** function to load the bitmaps from the Frntback.bmp file onto the surfaces. The **InitSurfaces** function uses the **DDCopyBitmap** function located in Ddutil.cpp to load both of the bitmaps, as shown in the following example:

```
// Load the bitmap resource.
hbm = (HBITMAP)LoadImage(GetModuleHandle(NULL), szBitmap,
    IMAGE_BITMAP, 0, 0, LR_CREATEDIBSECTION);

if (hbm == NULL)
    return FALSE;

DDCopyBitmap(lpDDSOne, hbm, 0, 0,   640, 480);
DDCopyBitmap(lpDDSTwo, hbm, 0, 480, 640, 480);
DeleteObject(hbm);

return TRUE;
```

If you look at the Frntback.bmp file in Microsoft Paint or another drawing application, you can see that the bitmap consists of two screens, one on top of the other. The **DDCopyBitmap** function breaks the bitmap in two at the point where the screens meet. In addition, it loads the first bitmap into the first off-screen surface (*lpDDSOne*) and the second bitmap into the second off-screen surface (*lpDDSTwo*).

Step 3: Blitting the Off-Screen Surfaces to the Back Buffer

The WM_TIMER message contains the code for writing to and flipping surfaces. In the case of DDEX3, it contains the following code to select the proper off-screen surface and to blit it to the back buffer:

```
rcRect.left = 0;
rcRect.top = 0;
rcRect.right = 640;
rcRect.bottom = 480;
if(phase)
{
    pdds = lpDDSTwo;
    phase = 0;
}
else
{
    pdds = lpDDSOne;
    phase = 1;
}
```

```
while(1)
{
    ddrval = lpDDSBack->BltFast(0, 0, pdds, &rcRect, FALSE);
    if(ddrval == DD_OK)
    {
        break;
    }
```

The phase variable determines which off-screen surface will be blitted to the back buffer. The **IDirectDrawSurface::BltFast** method is then called to blit the selected off-screen surface onto the back buffer, starting at position (0, 0), the upper-left corner. The *rcRect* parameter points to the **RECT** structure that defines the upper-left and lower-right corners of the off-screen surface that will be blitted from. The last parameter is set to FALSE (or 0), indicating that no specific transfer flags are used.

Depending on the requirements of your application, you could use either the **IDirectDrawSurface::Blt** method or the **IDirectDrawSurface::BltFast** method to blit from the off-screen buffer. If you are performing a blit from an off-screen plain buffer that is in display memory, you should use **IDirectDrawSurface::BltFast**. Although you will not gain speed on systems that use hardware blitters on their display adapters, the blit will take about 10 percent less time on systems that use hardware emulation to perform the blit. Because of this, you should use **IDirectDrawSurface::BltFast** for all display operations that blit from display memory to display memory. If you are blitting from system memory or require special hardware flags, however, you have to use **IDirectDrawSurface::Blt**.

After the off-screen surface is loaded in the back buffer, the back buffer and the primary surface are flipped in much the same way as shown in the previous tutorials.

Tutorial 4: Color Keys and Bitmap Animation

The sample in Tutorial 3 (DDEX3) shows a primitive method of placing bitmaps into an off-screen buffer before they are blitted to the back buffer. The sample in this tutorial (DDEX4) uses the techniques described in the previous tutorials to load a background and a series of sprites into an off-screen surface. Then it uses the **IDirectDrawSurface::BltFast** method to copy portions of the off-screen surface to the back buffer, thereby generating a simple bitmap animation.

The bitmap file that DDEX4 uses, All.bmp, contains the background and 60 iterations of a rotating red donut with a black background. The DDEX4 sample contains new functions that set the color key for the rotating donut sprites. Then, the sample copies the appropriate sprite to the back buffer from the off-screen surface.

The new functionality demonstrated in DDEX4 is shown in the following steps:

- *Step 1: Setting the Color Key*
- *Step 2: Creating a Simple Animation*

Step 1: Setting the Color Key

In addition to the other functions found in the **doInit** function of some of the other DirectDraw samples, the DDEX4 sample contains the code to set the color key for the sprites. Color keys are used for setting a color value that will be used for transparency. When the system contains a hardware blitter, all the pixels of a rectangle are blitted except the value that was set as the color key, thereby creating nonrectangular sprites on a surface. The code for setting the color key in DDEX4 is shown below:

```
// Set the color key for this bitmap (black).
DDSetColorKey(lpDDSOne, RGB(0,0,0));

return TRUE;
```

You can select the color key by setting the RGB values for the color you want in the call to the **DDSetColorKey** function. The RGB value for black is (0, 0, 0). The **DDSetColorKey** function calls the **DDColorMatch** function. (Both functions are in Ddutil.cpp.) The **DDColorMatch** function stores the current color value of the pixel at location (0, 0) on the bitmap located in the *lpDDSOne* surface. Then it takes the RGB values you supplied and sets the pixel at location (0, 0) to that color. Finally, it masks the value of the color with the number of bits per pixel that are available. After that is done, the original color is put back in location (0, 0), and the call returns to **DDSetColorKey** with the actual color key value. After it is returned, the color key value is placed in the **dwColorSpaceLowValue** member of the **DDCOLORKEY** structure. It is also copied to the **dwColorSpaceHighValue** member. The call to **IDirectDrawSurface::SetColorKey** then sets the color key.

You may have noticed the reference to CLR_INVALID in **DDSetColorKey** and **DDColorMatch**. If you pass CLR_INVALID as the color key in the **DDSetColorKey** call in DDEX4, the pixel in the upper-left corner (0, 0) of the bitmap will be used as the color key. As the DDEX4 bitmap is delivered, that does not mean much because the color of the pixel at (0, 0) is a shade of gray. If, however, you would like to see how to use the pixel at (0, 0) as the color key for the DDEX4 sample, open the All.bmp bitmap file in a drawing application and then change the single pixel at (0, 0) to black. Be sure to save the change (it's hard to see). Then change the DDEX4 line that calls **DDSetColorKey** to the following:

```
DDSetColorKey(lpDDSOne, CLR_INVALID);
```

Recompile the DDEX4 sample, and ensure that the resource definition file is also recompiled so that the new bitmap is included. (To do this, you can simply add and then delete a space in the Ddex4.rc file.) The DDEX4 sample will then use the pixel at (0, 0), which is now set to black, as the color key.

Step 2: Creating a Simple Animation

The DDEX4 sample uses the **updateFrame** function to create a simple animation using the red donuts included in the All.bmp file. The animation consists of three red donuts positioned in a triangle and rotating at various speeds. This sample compares the Win32 **GetTickCount** function with the number of milliseconds since the last call to **GetTickCount** to determine whether to redraw any of the sprites. It subsequently uses the **IDirectDrawSurface::BltFast** method first to blit the background from the off-screen surface (*lpDDSOne*) to the back buffer, and then to blit the sprites to the back buffer using the color key that you set earlier to determine which pixels are transparent. After the sprites are blitted to the back buffer, DDEX4 calls the **IDirectDrawSurface::Flip** method to flip the back buffer and the primary surface.

Note that when you use **IDirectDrawSurface::BltFast** to blit the background from the off-screen surface, the *dwTrans* parameter that specifies the type of transfer is set to DDBLTFAST_NOCOLORKEY. This indicates that a normal blit will occur with no transparency bits. Later, when the red donuts are blitted to the back buffer, the *dwTrans* parameter is set to DDBLTFAST_SRCCOLORKEY. This indicates that a blit will occur with the color key for transparency as it is defined, in this case, in the *lpDDSOne* buffer.

In this sample, the entire background is redrawn each time through the **updateFrame** function. One way of optimizing this sample would be to redraw only that portion of the background that changes while rotating the red donuts. Because the location and size of the rectangles that make up the donut sprites never change, you should be able to easily modify the DDEX4 sample with this optimization.

Tutorial 5: Dynamically Modifying Palettes

The sample described in this tutorial (DDEX5) is a modification of the sample described in Tutorial 4 (DDEX4) example. DDEX5 demonstrates how to dynamically change the palette entries while an application is running. The new functionality demonstrated in DDEX5 is shown in the following steps:

- *Step 1: Loading the Palette Entries*
- *Step 2: Rotating the Palettes*

Step 1: Loading the Palette Entries

The following code in DDEX5 loads the palette entries with the values in the lower half of the All.bmp file (the part of the bitmap that contains the red donuts):

```
// First, set all colors as unused.
for(i=0; i<256; i++)
{
    torusColors[i] = 0;
}

// Lock the surface and scan the lower part (the torus area),
// and keep track of all the indexes found.
ddsd.dwSize = sizeof(ddsd);
while (lpDDSOne->Lock(NULL, &ddsd, 0, NULL) == DDERR_WASSTILLDRAWING)
    ;

// Search through the torus frames and mark used colors.
for(y=480; y<480+384; y++)
{
    for(x=0; x<640; x++)
    {
        torusColors[((BYTE *)ddsd.lpSurface)[y*ddsd.lPitch+x]] = 1;
    }
}

lpDDSOne->Unlock(NULL);
```

The **torusColors** array is used as an indicator of the color index of the palette used in the lower half of the All.bmp file. Before it is used, all of the values in the **torusColors** array are reset to 0. The off-screen buffer is then locked in preparation for determining if a color index value is used.

The **torusColors** array is set to start at row 480 and column 0 of the bitmap. The color index value in the array is determined by the byte of data at the location in memory where the bitmap surface is located. This location is determined by the **lpSurface** member of the **DDSURFACEDESC** structure, which is pointing to the memory location corresponding to row 480 and column 0 of the bitmap ($y \times$ **lPitch** + x). The location of the specific color index value is then set to 1. The y value (row) is multiplied by the **lPitch** value (found in the **DDSURFACEDESC** structure) to get the actual location of the pixel in linear memory.

The color index values that are set in **torusColors** will be used later to determine which colors in the palette are rotated. Because there are no common colors between the background and the red donuts, only those colors associated with the red donuts are rotated. If you want to check whether this is true or not, just remove the "*ddsd.lPitch" from the array and see what happens when you recompile and run the program. (Without multiplying y$\times$**lPitch**, the red donuts

are never reached and only the colors found in the background are indexed and later rotated.) For more information about width and pitch, see *Width and Pitch*.

Step 2: Rotating the Palettes

The **updateFrame** function in DDEX5 works in much the same way as it did in Tutorial 4 (DDEX4). It first blits the background into the back buffer, and then it blits the three donuts in the foreground. However, before it flips the surfaces, **updateFrame** changes the palette of the primary surface from the palette index that was created in the **doInit** function, as shown in the following code:

```
// Change the palette.
if(lpDDPal->GetEntries(0, 0, 256, pe) != DD_OK)
{
    return;
}

for(i=1; i<256; i++)
{
    if(!torusColors[i])
    {
       continue;
    }
    pe[i].peRed = (pe[i].peRed+2) % 256;
    pe[i].peGreen = (pe[i].peGreen+1) % 256;
    pe[i].peBlue = (pe[i].peBlue+3) % 256;
}

if(lpDDPal->SetEntries(0, 0, 256, pe) != DD_OK)
{
    return;
}
```

The **IDirectDrawPalette::GetEntries** method in the first line queries palette values from a DirectDrawPalette object. Because the palette entry values pointed to by *pe* should be valid, the method will return DD_OK and continue. The loop that follows checks **torusColors** to determine if the color index was set to 1 during its initialization. If so, the red, green, and blue values in the palette entry pointed to by *pe* are rotated.

After all of the marked palette entries are rotated, the **IDirectDrawPalette::SetEntries** method is called to change the entries in the DirectDrawPalette object. This change takes place immediately if you are working with a palette set to the primary surface.

With this done, the surfaces are subsequently flipped DDEX5.

Other DirectDraw Samples

To learn more about how DirectDraw can be used in applications, you should check out some of the other following samples included with the DirectX SDK:

- Stretch

  Demonstrates how to create a nonexclusive (windowed) mode animation in a window that is capable of clipped blitting and stretched-clipped blitting.
- Donut

  Demonstrates testing multiple exclusive-mode applications interacting with nonexclusive-mode applications.
- Wormhole

  Demonstrates palette animation.
- Dxview

  Demonstrates how to retrieve the capabilities of the display hardware.

Other samples you can examine for their DirectDraw code include Duel, Iklowns, Foxbear, Palette, and Flip2d.

Optimizations and Customizations

All of the DirectDraw samples supplied with this SDK are relatively simple and assume a lot of things about the system they are running on. This section examines the following optimizations and customizations to the samples that will allow your code to work better in real-world situations:

- *Getting the Flip and Blit Status*
- *Blitting with Color Fill*
- *Determining the Capabilities of the Display Hardware*
- *Storing Bitmaps in Display Memory*
- *Triple Buffering*

Getting the Flip and Blit Status

When the **IDirectDrawSurface2::Flip** method is called, the primary surface and back buffer are exchanged. However, the exchange may not occur immediately. For example, if a previous flip has not finished, or if it did not succeed, this method returns DDERR_WASSTILLDRAWING. In the samples included with this SDK, the **IDirectDrawSurface2::Flip** call continues to loop until it returns DD_OK. Also, a **IDirectDrawSurface2::Flip** call does not complete immediately. It schedules a flip for the next time a vertical blank occurs on the system.

An application that waits until the DDERR_WASSTILLDRAWING value is not returned is very inefficient. Instead, you could create a function in your application that calls the **IDirectDrawSurface2::GetFlipStatus** method on the back buffer to determine if the previous flip has finished.

If the previous flip has not finished and the call returns DDERR_WASSTILLDRAWING, your application can use the time to perform another task before it checks the status again. Otherwise, you can perform the next flip. The following example demonstrates this concept:

```
while(lpDDSBack->GetFlipStatus(DDGFS_ISFLIPDONE) ==
    DDERR_WASSTILLDRAWING);

    // Waiting for the previous flip to finish. The application can
    // perform another task here.

ddrval = lpDDSPrimary->Flip(NULL, 0);
```

You can use the **IDirectDrawSurface2::GetBltStatus** method in much the same way to determine whether a blit has finished. Because **IDirectDrawSurface2::GetFlipStatus** and **IDirectDrawSurface2::GetBltStatus** return immediately, you can use them periodically in your application with little loss in speed.

Blitting with Color Fill

You can use the **IDirectDrawSurface2::Blt** method to perform a color fill of the most common color you want to be displayed. For example, if the most common color your application displays is blue, you can use **IDirectDrawSurface2::Blt** with the DDBLT_COLORFILL flag to first fill the surface with the color blue. Then you can write everything else on top of it. This allows you to fill in the most common color very quickly, and you then only have to write a minimum number of colors to the surface.

The following example demonstrates one way to perform a color fill:

```
DDBLTFX ddbltfx;

ddbltfx.dwSize = sizeof(ddbltfx);
ddbltfx.dwFillColor = 0;
ddrval = lpDDSPrimary->Blt(
    NULL,        // Destination
    NULL, NULL,  // Source rectangle
    DDBLT_COLORFILL, &ddbltfx);

switch(ddrval)
{
    case DDERR_WASSTILLDRAWING:
        .
```

```
        .
        .
    case DDERR_SURFACELOST:
        .
        .
        .
    case DD_OK:
        .
        .
        .
    default:
}
```

Determining the Capabilities of the Display Hardware

DirectDraw uses hardware emulation to perform the DirectDraw functions not supported by the user's hardware. To accelerate performance of your DirectDraw applications, you should determine the capabilities of the user's display hardware after you have created a DirectDraw object. DirectDraw will use any display acceleration hardware available on the user's system. Note that your application must supply DirectDraw with a list of the hardware emulation it needs in case the display adapter on the user's system does not contain the display acceleration hardware required by your application.

You can use the **IDirectDraw2::GetCaps** method to fill in the capabilities of the display hardware. The DirectDraw device driver for the hardware fills in the values of the **dwCaps** member of the **DDCAPS** structure. These values identify the capabilities of the display acceleration hardware on the system. The **DDCAPS** structure contains the address of the **DDSCAPS** structure that supplies hardware emulation requirements for the application. Hardware emulation will be used in case any or all of the DirectDraw hardware capabilities are not available on the display adapter. You must supply the hardware emulation values your application requires in the **DDSCAPS** structure.

Storing Bitmaps in Display Memory

Blitting from display memory to display memory is usually much more efficient than blitting from system memory to display memory. As a result, you should store as many of the sprites your application uses as possible in display memory.

Most display adapter hardware contains enough extra memory to store more than only the primary surface and the back buffer. You can use the **dwVidMemTotal** and **dwVidMemFree** members of the **DDCAPS** structure (if you used the **IDirectDraw2::GetCaps** method to get the capabilities of the user's display hardware) to determine the amount of available memory for storing bitmaps in the display adapter's memory. If you want to see how this works, use the DirectX Viewer application supplied with the DirectX SDK. Under DirectDraw Devices,

open the Primary Display Driver folder, and then open the General folder. The amount of total display memory (minus the primary surface) and the amount of free memory is displayed. Each time a surface is added to the DirectDraw object, the amount of free memory decreases by the amount of memory used by the added surface.

Triple Buffering

In some cases, it may be possible to speed up the process of displaying your application by using triple buffering. Triple buffering uses one primary surface and two back buffers. The following example shows how to initialize a triple-buffering scheme:

```
// Create the primary surface with two back buffers.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
    DDSCAPS_FLIP | DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 2;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
if(ddrval == DD_OK)
{
    // Get a pointer to the first back buffer.
    ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
    ddrval = lpDDSPrimary->GetAttachedSurface(&ddscaps,
        &lpDDSBackOne);
    if(ddrval != DD_OK)
        // Display an error message here.
    // Retrieve a pointer to the second back buffer.
    ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
    ddrval = lpDDSPrimary->GetAttachedSurface(&ddscaps,
        &lpDDSBackTwo);
```

Triple buffering allows your application to continue blitting to a back buffer even if a flip has not completed and the first back buffer's blit has already finished. Performing a flip is not a synchronous event; one flip can take longer than another. Therefore, if your application uses only one back buffer, it may spend some time idling while waiting for the **IDirectDrawSurface2::Flip** method to return with DD_OK.

DirectDraw Reference

Functions

DirectDrawCreate

```
HRESULT DirectDrawCreate(GUID FAR * lpGUID,
```

```
    LPDIRECTDRAW FAR * lplpDD, IUnknown FAR * pUnkOuter);
```

Creates an instance of a DirectDraw object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_DIRECTDRAWALREADYCREATED

  DDERR_GENERIC

  DDERR_INVALIDDIRECTDRAWGUID

  DDERR_INVALIDPARAMS

  DDERR_NODIRECTDRAWHW

  DDERR_OUTOFMEMORY

lpGUID
Address of the globally unique identifier (GUID) that represents the driver to be created. NULL is always the active display driver.

lplpDD
Address of a pointer that will be initialized with a valid DirectDraw pointer if the call succeeds.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, **DirectDrawCreate** returns an error if this parameter is anything but NULL.

This function attempts to initialize a DirectDraw object, and it then sets a pointer to the object if the call is successful. Calling the **IDirectDraw2::GetCaps** method immediately after initialization is advised to determine to what extent this object is hardware accelerated.

DirectDrawCreateClipper

```
HRESULT DirectDrawCreateClipper(DWORD dwFlags,
    LPDIRECTDRAWCLIPPER FAR *lplpDDClipper,
    IUnknown FAR *pUnkOuter);
```

Creates an instance of a DirectDrawClipper object not associated with a DirectDraw object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDPARAMS

  DDERR_OUTOFMEMORY

dwFlags
This parameter is currently not used and must be set to 0.

lplpDDClipper
Address of a pointer that will be filled with the address of the new DirectDrawClipper object.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, **DirectDrawCreateClipper** returns an error if this parameter is anything but NULL.

This function can be called before any DirectDraw objects are created. Because these DirectDrawClipper objects are not owned by any DirectDraw object, they are not automatically released when an application's objects are released. If the application does not explicitly release the DirectDrawClipper objects, DirectDraw will release them when the application terminates.

To create a DirectDrawClipper object owned by a specific DirectDraw object, use the **IDirectDraw2::CreateClipper** method.

See also **IDirectDraw2::CreateClipper**

DirectDrawEnumerate

```
HRESULT DirectDrawEnumerate(LPDDENUMCALLBACK lpCallback,
    LPVOID lpContext);
```

Enumerates the DirectDraw objects installed on the system. The NULL GUID entry always identifies the primary display device shared with GDI.

- Returns DD_OK if successful, or **DDERR_INVALIDPARAMS** otherwise.

lpCallback
Address of a **Callback** function that will be called with a description of each DirectDraw-enabled HAL installed in the system.

lpContext
Address of an application-defined context that will be passed to the enumeration callback function each time it is called.

Callback Functions

Callback

```
BOOL WINAPI lpCallback(GUID FAR * lpGUID,
    LPSTR lpDriverDescription, LPSTR lpDriverName,
    LPVOID lpContext);
```

Application-defined callback function for the **DirectDrawEnumerate** function.

- Returns DDENUMRET_OK to continue the enumeration, or DDENUMRET_CANCEL to stop it.

lpGUID
Address of the unique identifier of the DirectDraw object.

lpDriverDescription
Address of a string containing the driver description.

lpDriverName
Address of a string containing the driver name.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

EnumModesCallback

```
HRESULT WINAPI lpEnumModesCallback(LPDDSURFACEDESC lpDDSurfaceDesc,
    LPVOID lpContext);
```

Application-defined callback function for the **IDirectDraw2::EnumDisplayModes** method.

- Returns DDENUMRET_OK to continue the enumeration, or DDENUMRET_CANCEL to stop it.

lpDDSurfaceDesc
Address of the **DDSURFACEDESC** structure that provides the monitor frequency and the mode that can be created. This data is read-only.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

EnumSurfacesCallback

```
HRESULT WINAPI lpEnumSurfacesCallback(
    LPDIRECTDRAWSURFACE2 lpDDSurface,
    LPDDSURFACEDESC lpDDSurfaceDesc, LPVOID lpContext);
```

Application-defined callback function for the **IDirectDrawSurface2::EnumAttachedSurfaces** method.

- Returns DDENUMRET_OK to continue the enumeration, or DDENUMRET_CANCEL to stop it.

lpDDSurface
Address of the surface attached to this surface.

lpDDSurfaceDesc
Address of a **DDSURFACEDESC** structure that describes the attached surface.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

fnCallback

```
HRESULT WINAPI lpfnCallback(LPDIRECTDRAWSURFACE lpDDSurface,
    LPVOID lpContext);
```

Application-defined callback function for the **IDirectDrawSurface2::EnumOverlayZOrders** method.

- Returns DDENUMRET_OK to continue the enumeration, or DDENUMRET_CANCEL to stop it.

lpDDSurface
Address of the surface being overlaid on this surface.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

IDirectDraw2

Applications use the methods of the **IDirectDraw2** interface to create DirectDraw objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see *DirectDraw Objects*.

The methods of the **IDirectDraw2** interface can be organized into the following groups:

Allocating memory	**Compact**
	Initialize
Creating objects	**CreateClipper**
	CreatePalette
	CreateSurface
Device capabilities	**GetCaps**
Display modes	**EnumDisplayModes**
	GetDisplayMode
	GetMonitorFrequency
	RestoreDisplayMode

	SetDisplayMode
Display status	**GetScanLine** **GetVerticalBlankStatus**
Miscellaneous	**GetAvailableVidMem** **GetFourCCCodes** **WaitForVerticalBlank**
Setting behavior	**SetCooperativeLevel**
Surfaces	**DuplicateSurface** **EnumSurfaces** **FlipToGDISurface** **GetGDISurface**

The **IDirectDraw2** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

IDirectDraw2::Compact

```
HRESULT Compact();
```

At present this method is only a stub; it has not yet been implemented.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOEXCLUSIVEMODE

  DDERR_SURFACEBUSY

This method moves all of the pieces of surface memory on the display card to a contiguous block to make the largest single amount of free memory available. This call fails if any operations are in progress.

The application calling this method must have its cooperative level set to exclusive.

IDirectDraw2::CreateClipper

```
HRESULT CreateClipper(DWORD dwFlags,
    LPDIRECTDRAWCLIPPER FAR * lplpDDClipper,
    IUnknown FAR * pUnkOuter);
```

Creates a DirectDrawClipper object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOCOOPERATIVELEVELSET

  DDERR_OUTOFMEMORY

dwFlags
This parameter is currently not used and must be set to 0.

lplpDDClipper
Address of a pointer that will be filled with the address of the new DirectDrawClipper object if this method returns successfully.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, **IDirectDraw2::CreateClipper** returns an error if this parameter is anything but NULL.

The DirectDrawClipper object can be attached to a DirectDrawSurface and used during **IDirectDrawSurface2::Blt**, **IDirectDrawSurface2::BltBatch**, and **IDirectDrawSurface2::UpdateOverlay** operations.

To create a DirectDrawClipper object that is not owned by a specific DirectDraw object, use the **DirectDrawCreateClipper** function.

See also **IDirectDrawSurface2::GetClipper**, **IDirectDrawSurface2::SetClipper**

IDirectDraw2::CreatePalette

```
HRESULT CreatePalette(DWORD dwFlags,
    LPPALETTEENTRY lpColorTable,
    LPDIRECTDRAWPALETTE FAR * lplpDDPalette,
    IUnknown FAR * pUnkOuter);
```

Creates a DirectDrawPalette object for this DirectDraw object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOCOOPERATIVELEVELSET

  DDERR_NOEXCLUSIVEMODE

  DDERR_OUTOFCAPS

  DDERR_OUTOFMEMORY

  DDERR_UNSUPPORTED

dwFlags

One or more of the following flags:

DDPCAPS_1BIT

Indicates that the index is 1 bit. There are two entries in the color table.

DDPCAPS_2BIT

Indicates that the index is 2 bits. There are four entries in the color table.

DDPCAPS_4BIT

Indicates that the index is 4 bits. There are 16 entries in the color table.

DDPCAPS_8BITENTRIES

Indicates that the index refers to an 8-bit color index. This flag is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT flag, and when the target surface is in 8 bpp. Each color entry is 1 byte long and is an index to a destination surface's 8-bpp palette.

DDPCAPS_8BIT

Indicates that the index is 8 bits. There are 256 entries in the color table.

DDPCAPS_ALLOW256

Indicates that this palette can have all 256 entries defined.

lpColorTable

Address of an array of 2, 4, 16, or 256 **PALETTEENTRY** structures that will initialize this DirectDrawPalette object.

lplpDDPalette

Address of a pointer that will be filled with the address of the new DirectDrawPalette object if this method returns successfully.

pUnkOuter

Allows for future compatibility with COM aggregation features. Presently, however, **IDirectDraw2::CreatePalette** returns an error if this parameter is anything but NULL.

IDirectDraw2::CreateSurface

```
HRESULT CreateSurface(LPDDSURFACEDESC lpDDSurfaceDesc,
    LPDIRECTDRAWSURFACE FAR * lplpDDSurface,
    IUnknown FAR * pUnkOuter);
```

Creates a DirectDrawSurface object for this DirectDraw object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INCOMPATIBLEPRIMARY

  DDERR_INVALIDCAPS

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDPIXELFORMAT

  DDERR_NOALPHAHW

  DDERR_NOCOOPERATIVELEVELSET

  DDERR_NODIRECTDRAWHW

  DDERR_NOEMULATION

  DDERR_NOEXCLUSIVEMODE

  DDERR_NOFLIPHW

  DDERR_NOMIPMAPHW

  DDERR_NOZBUFFERHW

  DDERR_OUTOFMEMORY

  DDERR_OUTOFVIDEOMEMORY

  DDERR_PRIMARYSURFACEALREADYEXISTS

  DDERR_UNSUPPORTEDMODE

lpDDSurfaceDesc
Address of the **DDSURFACEDESC** structure that describes the requested surface.

lplpDDSurface
Address of a pointer to be initialized with a valid DirectDrawSurface pointer if the call succeeds.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, **IDirectDraw2::CreateSurface** returns an error if this parameter is anything but NULL.

IDirectDraw2::DuplicateSurface

```
HRESULT DuplicateSurface(LPDIRECTDRAWSURFACE lpDDSurface,
    LPLPDIRECTDRAWSURFACE FAR * lplpDupDDSurface);
```

Duplicates a DirectDrawSurface object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_CANTDUPLICATE

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_OUTOFMEMORY

  DDERR_SURFACELOST

lpDDSurface
Address of the DirectDrawSurface structure to be duplicated.

lplpDupDDSurface
Address of the DirectDrawSurface pointer that points to the newly created duplicate DirectDrawSurface structure.

This method creates a new DirectDrawSurface object that points to the same surface memory as an existing DirectDrawSurface object. This duplicate can be used just like the original object. The surface memory is released after the last object referencing it is released. A primary surface, 3D surface, or implicitly created surface cannot be duplicated.

IDirectDraw2::EnumDisplayModes

```
HRESULT EnumDisplayModes(DWORD dwFlags,
    LPDDSURFACEDESC lpDDSurfaceDesc, LPVOID lpContext,
    LPDDENUMMODESCALLBACK lpEnumModesCallback);
```

Enumerates all of the display modes the hardware exposes through the DirectDraw object that are compatible with a provided surface description. If NULL is passed for the surface description, all exposed modes are enumerated.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

dwFlags

DDEDM_REFRESHRATES

Enumerates modes with different refresh rates. **IDirectDraw2::EnumDisplayModes** guarantees that a particular mode will be enumerated only once. This flag specifies whether the refresh rate is taken into account when determining if a mode is unique.

lpDDSurfaceDesc
Address of a **DDSURFACEDESC** structure that will be checked against available modes. If the value of this parameter is NULL, all modes are enumerated.

lpContext
Address of an application-defined structure that will be passed to each enumeration member.

lpEnumModesCallback
Address of the **EnumModesCallback** function that the enumeration procedure will call every time a match is found.

This method enumerates the **dwRefreshRate** member of the **DDSURFACEDESC** structure; the **IDirectDraw::EnumDisplayModes** method does not contain this capability. If you use the **IDirectDraw2::SetDisplayMode** method to set the refresh rate of a new mode, you must use **IDirectDraw2::EnumDisplayModes** to enumerate the **dwRefreshRate** member.

See also **IDirectDraw2::GetDisplayMode**, **IDirectDraw2::SetDisplayMode**, **IDirectDraw2::RestoreDisplayMode**

IDirectDraw2::EnumSurfaces

```
HRESULT EnumSurfaces(DWORD dwFlags, LPDDSURFACEDESC lpDDSD,
    LPVOID lpContext, LPDDENUMSURFACESCALLBACK lpEnumSurfacesCallback);
```

Enumerates all of the existing or possible surfaces that meet the search criterion specified.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

dwFlags
One of the following flags:

DDENUMSURFACES_ALL

Enumerates all of the surfaces that meet the search criterion.

DDENUMSURFACES_CANBECREATED

Enumerates the first surface that can be created and meets the search criterion.

DDENUMSURFACES_DOESEXIST

Enumerates the already existing surfaces that meet the search criterion.

DDENUMSURFACES_MATCH

Searches for any surface that matches the surface description.

DDENUMSURFACES_NOMATCH

Searches for any surface that does not match the surface description.

lpDDSD
Address of a **DDSURFACEDESC** structure that defines the surface of interest.

lpContext
Address of an application-defined structure that will be passed to each enumeration member.

lpEnumSurfacesCallback
Address of the **EnumSurfacesCallback** function the enumeration procedure will call every time a match is found.

If the DDENUMSURFACES_CANBECREATED flag is set, this method attempts to temporarily create a surface that meets the criterion. Note that as a surface is enumerated, its reference count is increased—if you are not going to use the surface, use **IDirectDraw::Release** to release the surface after each enumeration.

As part of the **IDirectDraw** interface, this method did not support any values other than zero for the *dwFlags* parameter.

IDirectDraw2::FlipToGDISurface

```
HRESULT FlipToGDISurface();
```

Makes the surface that GDI writes to the primary surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOTFOUND

This method can be called at the end of a page-flipping application to ensure that the display memory that GDI is writing to is visible to the user.

See also **IDirectDraw2::GetGDISurface**

IDirectDraw2::GetAvailableVidMem

```
HRESULT GetAvailableVidMem(LPDDSCAPS lpDDSCaps,
    LPDWORD lpdwTotal, LPDWORD lpdwFree);
```

Retrieves the total amount of display memory available and the amount of display memory currently free.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDCAPS

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NODIRECTDRAWHW

lpDDSCaps
Address of a **DDSCAPS** structure that contains the hardware capabilities of the surface.

lpdwTotal
Address of a variable that will be filled with the total amount of display memory available.

lpdwFree
Address of a variable that will be filled with the amount of display memory currently free.

If NULL is passed to either *lpdwTotal* or *lpdwFree*, the value for that parameter is not returned.

The following C++ example demonstrates using **IDirectDraw2::GetAvailableVidMem** to determine both the total and free display memory available for texture-map surfaces:

```
LPDIRECTDRAW2 lpDD2;
DDSCAPS       ddsCaps;
DWORD         dwTotal;
DWORD         dwFree;

ddres = lpDD->QueryInterface(IID_IDirectDraw2, &lpDD2);
if (FAILED(ddres))
.
.
.
ddsCaps.dwCaps = DDSCAPS_TEXTURE;
ddres = lpDD2->GetAvailableVidMem(&ddsCaps, &dwTotal, &dwFree);
if (FAILED(ddres))
.
.
.
```

This method provides only a snapshot of the current display-memory state. The amount of free display memory is subject to change as surfaces are created and released. Therefore, you should use the free memory value only as an approximation. In addition, a particular display adapter card may make no distinction between two different memory types. For example, the adapter might use the same portion of display memory to store z-buffers and textures. So,

allocating one type of surface (for example, a z-buffer) can affect the amount of display memory available for another type of surface (for example, textures). Therefore, it is best to first allocate an application's fixed resources (such as front and back buffers, and z-buffers) before determining how much memory is available for dynamic use (such as texture mapping).

This method was not implemented in the **IDirectDraw** interface.

IDirectDraw2::GetCaps

```
HRESULT GetCaps(LPDDCAPS lpDDDriverCaps, LPDDCAPS lpDDHELCaps);
```

Fills in the capabilities of the device driver for the hardware and the hardware emulation layer (HEL).

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lpDDDriverCaps
Address of a **DDCAPS** structure that will be filled with the capabilities of the hardware, as reported by the device driver.

lpDDHELCaps
Address of a **DDCAPS** structure that will be filled with the capabilities of the HEL.

See also **DDCAPS**

IDirectDraw2::GetDisplayMode

```
HRESULT GetDisplayMode(LPDDSURFACEDESC lpDDSurfaceDesc);
```

Retrieves the current display mode.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_UNSUPPORTEDMODE

lpDDSurfaceDesc
Address of a **DDSURFACEDESC** structure that will be filled with a description of the surface.

An application should not save the information returned by this method to restore the display mode on clean-up. The application should use the **IDirectDraw2::RestoreDisplayMode** method to restore the mode on clean-up,

thereby avoiding mode-setting conflicts that could arise in a multiprocess environment.

See also **IDirectDraw2::SetDisplayMode**, **IDirectDraw2::RestoreDisplayMode**, **IDirectDraw2::EnumDisplayModes**

IDirectDraw2::GetFourCCCodes

```
HRESULT GetFourCCCodes(LPDWORD lpNumCodes, LPDWORD lpCodes);
```

Retrieves the FourCC codes supported by the DirectDraw object. This method can also retrieve the number of codes supported.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lpNumCodes
: Address of a variable that contains the number of entries the array pointed to by *lpCodes* can hold. If the number of entries is too small to accommodate all the codes, *lpNumCodes* is set to the required number and the array pointed to by *lpCodes* is filled with all that fits.

lpCodes
: Address of an array of variables that will be filled with FourCC codes supported by this DirectDraw object. If NULL is passed, *lpNumCodes* is set to the number of supported FourCC codes and the method will return.

IDirectDraw2::GetGDISurface

```
HRESULT GetGDISurface(LPDIRECTDRAWSURFACE FAR * lplpGDIDDSSurface);
```

Retrieves the DirectDrawSurface object that currently represents the surface memory that GDI is treating as the primary surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOTFOUND

lplpGDIDDSSurface
: Address of a DirectDrawSurface pointer to the DirectDrawSurface object that currently controls GDI's primary surface memory.

See also **IDirectDraw2::FlipToGDISurface**

IDirectDraw2::GetMonitorFrequency

```
HRESULT GetMonitorFrequency(LPDWORD lpdwFrequency);
```

Retrieves the frequency of the monitor being driven by the DirectDraw object. The frequency value is returned in Hz multiplied by 100. For example, 60Hz is returned as 6000.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_UNSUPPORTED

lpdwFrequency
Address of the variable that will be filled with the monitor frequency.

IDirectDraw2::GetScanLine

```
HRESULT GetScanLine(LPDWORD lpdwScanLine);
```

Retrieves the scanline that is currently being drawn on the monitor.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_UNSUPPORTED

  DDERR_VERTICALBLANKINPROGRESS

lpdwScanLine
Address of the variable that will contain the scanline the display is currently on.

See also **IDirectDraw2::GetVerticalBlankStatus**, **IDirectDraw2::WaitForVerticalBlank**

IDirectDraw2::GetVerticalBlankStatus

```
HRESULT GetVerticalBlankStatus(LPBOOL lpbIsInVB);
```

Retrieves the status of the vertical blank.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lpbIsInVB
Address of the variable that will be filled with the status of the vertical blank. This parameter is TRUE if a vertical blank is occurring, and FALSE otherwise.

To synchronize with the vertical blank, use the **IDirectDraw2::WaitForVerticalBlank** method.

See also **IDirectDraw2::GetScanLine**, **IDirectDraw2::WaitForVerticalBlank**

IDirectDraw2::Initialize

```
HRESULT Initialize(GUID FAR * lpGUID);
```

Initializes the DirectDraw object that was created by using the **CoCreateInstance** OLE function.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_ALREADYINITIALIZED

  DDERR_DIRECTDRAWALREADYCREATED

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NODIRECTDRAWHW

  DDERR_NODIRECTDRAWSUPPORT

  DDERR_OUTOFMEMORY

lpGUID
Address of the globally unique identifier (GUID) used as the interface identifier.

This method is provided for compliance with the Component Object Model (COM) protocol. If the **DirectDrawCreate** function was used to create the DirectDraw object, this method returns DDERR_ALREADYINITIALIZED. If **IDirectDraw2::Initialize** is not called when using **CoCreateInstance** to create the DirectDraw object, any method that is called afterward returns DDERR_NOTINITIALIZED.

For more information about using **IDirectDraw2::Initialize** with **CoCreateInstance**, see *Creating DirectDraw Objects by Using CoCreateInstance*.

See also **IUnknown::AddRef**, **IUnknown::QueryInterface**, **IUnknown::Release**

IDirectDraw2::RestoreDisplayMode

```
HRESULT RestoreDisplayMode();
```

Resets the mode of the display device hardware for the primary surface to what it was before the **IDirectDraw2::SetDisplayMode** method was called. Exclusive-level access is required to use this method.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_LOCKEDSURFACES

  DDERR_NOEXCLUSIVEMODE

See also **IDirectDraw2::SetDisplayMode**, **IDirectDraw2::EnumDisplayModes**, **IDirectDraw2::SetCooperativeLevel**

IDirectDraw2::SetCooperativeLevel

```
HRESULT SetCooperativeLevel(HWND hWnd, DWORD dwFlags);
```

Determines the top-level behavior of the application.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_EXCLUSIVEMODEALREADYSET

  DDERR_HWNDALREADYSET

  DDERR_HWNDSUBCLASSED

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_OUTOFMEMORY

hWnd
Window handle used for the application.

dwFlags
One or more of the following flags:

DDSCL_ALLOWMODEX
Allows the use of Mode X display modes.

DDSCL_ALLOWREBOOT
Allows CTRL+ALT+DEL to function while in exclusive (full-screen) mode.

DDSCL_EXCLUSIVE
Requests the exclusive level.

DDSCL_FULLSCREEN

Indicates that the exclusive-mode owner will be responsible for the entire primary surface. GDI can be ignored.

DDSCL_NORMAL

Indicates that the application will function as a regular Windows application.

DDSCL_NOWINDOWCHANGES

Indicates that DirectDraw is not allowed to minimize or restore the application window on activation.

The DDSCL_EXCLUSIVE flag must be set to call functions that can have drastic performance consequences for other applications. To call the **IDirectDraw2::Compact** method, change the display mode, or modify the behavior (for example, flipping) of the primary surface, an application must be set to the exclusive level. If an application calls the **IDirectDraw2::SetCooperativeLevel** method with the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags set, DirectDraw will attempt to resize its window to full screen. An application must set either the DDSCL_EXCLUSIVE or DDSCL_NORMAL flag, and DDSCL_EXCLUSIVE requires DDSCL_FULLSCREEN.

Mode X modes are available only if an application sets the DDSCL_ALLOWMODEX, DDSCL_FULLSCREEN, and DDSCL_EXCLUSIVE flags. DDSCL_ALLOWMODEX cannot be used with DDSCL_NORMAL. If DDSCL_ALLOWMODEX is not specified, the **IDirectDraw2::EnumDisplayModes** method will not enumerate the Mode X modes, and the **IDirectDraw2::SetDisplayMode** method will fail when a Mode X mode is requested. The set of supported display modes may change after using **IDirectDraw2::SetCooperativeLevel**.

Windows does not support Mode X modes; therefore, when your application is in a Mode X mode, you cannot use the **IDirectDrawSurface2::Lock** or **IDirectDrawSurface2::Blt** methods to lock or blit the primary surface. You also cannot use either the **IDirectDrawSurface2::GetDC** method on the primary surface, or GDI with a screen DC. Mode X modes are indicated by the DDSCAPS_MODEX flag in the **DDSCAPS** structure, which is part of the **DDSURFACEDESC** structure returned by the **IDirectDrawSurface2::GetCaps** and **IDirectDraw2::EnumDisplayModes** methods.

Because applications can use DirectDraw with multiple windows, **IDirectDraw2::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.

Interaction between the **IDirectDraw::SetDisplayMode** and **IDirectDraw::SetCooperativeLevel** methods differs from their *IDirectDraw2*

counterparts. That is, if an application uses the **IDirectDraw** versions of these interfaces to set the cooperative level and display modes according to the following steps, the original display mode must be restored by using the **IDirectDraw::RestoreDisplayMode** method.

1 Call **IDirectDraw::SetCooperativeLevel** with the DDSCL_EXCLUSIVE flag to gain exclusive mode.
2 Call **IDirectDraw::SetDisplayMode** to change the display mode.
3 Call **IDirectDraw::SetCooperativeLevel** with the DDSCL_NORMAL flag to release exclusive mode.

However, if you use the **IDirectDraw2** interface and follow the same steps, the original display mode will be restored when exclusive mode is lost.

See also **IDirectDraw2::SetDisplayMode**, **IDirectDraw2::Compact**, **IDirectDraw2::EnumDisplayModes**

IDirectDraw2::SetDisplayMode

```
HRESULT SetDisplayMode(DWORD dwWidth, DWORD dwHeight,
    DWORD dwBPP, DWORD dwRefreshRate, DWORD dwFlags);
```

Sets the mode of the display-device hardware.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC
  DDERR_INVALIDMODE
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS
  DDERR_LOCKEDSURFACES
  DDERR_NOEXCLUSIVEMODE
  DDERR_SURFACEBUSY
  DDERR_UNSUPPORTED
  DDERR_UNSUPPORTEDMODE
  DDERR_WASSTILLDRAWING

dwWidth and *dwHeight*
Width and height of the new mode.

dwBPP
Bits per pixel (bpp) of the new mode.

dwRefreshRate
Refresh rate of the new mode. If this parameter is set to 0, the **IDirectDraw** interface version of this method is used.

dwFlags
This parameter is currently not used and must be set to 0.

The **IDirectDraw2::SetCooperativeLevel** method must be used to set exclusive-level access before the mode can be changed. If other applications have created a DirectDrawSurface object on the primary surface and the mode is changed, those applications' primary surface objects return DDERR_SURFACELOST until they are restored.

As part of the **IDirectDraw** interface, this method did not include the *dwRefreshRate* and *dwFlags* parameters.

Interaction between the **IDirectDraw::SetDisplayMode** and **IDirectDraw::SetCooperativeLevel** methods differs from their *IDirectDraw2* counterparts. That is, if an application uses the **IDirectDraw** versions of these interfaces to set the cooperative level and display modes according to the following steps, the original display mode must be restored by using the **IDirectDraw::RestoreDisplayMode** method.

1. Call **IDirectDraw::SetCooperativeLevel** with the DDSCL_EXCLUSIVE flag to gain exclusive mode.
2. Call **IDirectDraw::SetDisplayMode** to change the display mode.
3. Call **IDirectDraw::SetCooperativeLevel** with the DDSCL_NORMAL flag to release exclusive mode.

However, if you use the **IDirectDraw2** interface and follow the same steps, the original display mode will be restored when exclusive mode is lost.

See also **IDirectDraw2::RestoreDisplayMode**, **IDirectDraw2::GetDisplayMode**, **IDirectDraw2::EnumDisplayModes**, **IDirectDraw2::SetCooperativeLevel**

IDirectDraw2::WaitForVerticalBlank

```
HRESULT WaitForVerticalBlank(DWORD dwFlags, HANDLE hEvent);
```

Helps the application synchronize itself with the vertical-blank interval.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_UNSUPPORTED

  DDERR_WASSTILLDRAWING

dwFlags
Determines how long to wait for the vertical blank.

DDWAITVB_BLOCKBEGIN

Returns when the vertical-blank interval begins.

DDWAITVB_BLOCKBEGINEVENT

Triggers an event when the vertical blank begins. This value is not currently supported.

DDWAITVB_BLOCKEND

Returns when the vertical-blank interval ends and the display begins.

hEvent

Handle of the event to be triggered when the vertical blank begins.

See also **IDirectDraw2::GetVerticalBlankStatus**, **IDirectDraw2::GetScanLine**

IDirectDrawClipper

Applications use the methods of the **IDirectDrawClipper** interface to manage clip lists. This section is a reference to the methods of this interface. For a conceptual overview, see *IDirectDrawClipper Interface*.

The methods of the **IDirectDrawClipper** interface can be organized into the following groups:

Allocating memory	**Initialize**
Clip lists	**GetClipList**
	IsClipListChanged
	SetClipList
	SetHWnd
Handles	**GetHWnd**

The **IDirectDrawClipper** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

IDirectDrawClipper::GetClipList

```
HRESULT GetClipList(LPRECT lpRect, LPRGNDATA lpClipList,
```

```
    LPDWORD lpdwSize);
```

Retrieves a copy of the clip list associated with a DirectDrawClipper object. A subset of the clip list can be selected by passing a rectangle that clips the clip list.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDCLIPLIST

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOCLIPLIST

  DDERR_REGIONTOOSMALL

lpRect
Address of a rectangle that will be used to clip the clip list.

lpClipList
Address of an **RGNDATA** structure that will contain the resulting copy of the clip list.

lpdwSize
Size of the resulting clip list.

The **RGNDATA** structure used with this method has the following syntax.

```
typedef struct _RGNDATA {
    RGNDATAHEADER rdh;
    char          Buffer[1];
} RGNDATA;
```

The **rdh** member of the **RGNDATA** structure is an **RGNDATAHEADER** structure that has the following syntax.

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize;
    DWORD iType;
    DWORD nCount;
    DWORD nRgnSize;
    RECT  rcBound;
} RGNDATAHEADER;
```

For more information about these structures, see the documentation in the Win32 Software Development Kit.

See also **IDirectDrawClipper::SetClipList**

IDirectDrawClipper::GetHWnd

```
HRESULT GetHWnd(HWND FAR * lphWnd);
```

Retrieves the window handle previously associated with this DirectDrawClipper object by the **IDirectDrawClipper::SetHWnd** method.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lphWnd
Address of the window handle previously associated with this DirectDrawClipper object by the **IDirectDrawClipper::SetHWnd** method.

See also **IDirectDrawClipper::SetHWnd**

IDirectDrawClipper::Initialize

```
HRESULT Initialize(LPDIRECTDRAW lpDD, DWORD dwFlags);
```

Initializes a DirectDrawClipper object that was created by using the **CoCreateInstance** OLE function.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_ALREADYINITIALIZED

  DDERR_INVALIDPARAMS

lpDD
Address of the DirectDraw structure that represents the DirectDraw object. If this parameter is set to NULL, an independent DirectDrawClipper object is created (the equivalent of using the **DirectDrawCreateClipper** function).

dwFlags
This parameter is currently not used and must be set to 0.

This method is provided for compliance with the Component Object Model (COM) protocol. If **DirectDrawCreateClipper** or the **IDirectDraw2::CreateClipper** method was used to create the DirectDrawClipper object, this method returns DDERR_ALREADYINITIALIZED.

For more information about using **IDirectDrawClipper::Initialize** with **CoCreateInstance**, see *Creating DirectDrawClipper Objects with CoCreateInstance*.

See also **IUnknown::AddRef**, **IUnknown::QueryInterface**, **IUnknown::Release**, **IDirectDraw2::CreateClipper**

IDirectDrawClipper::IsClipListChanged

```
HRESULT IsClipListChanged(BOOL FAR * lpbChanged);
```

Monitors the status of the clip list if a window handle is associated with a DirectDrawClipper object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lpbChanged
Address of a variable that is set to TRUE if the clip list has changed.

IDirectDrawClipper::SetClipList

```
HRESULT SetClipList(LPRGNDATA lpClipList, DWORD dwFlags);
```

Sets or deletes the clip list used by the **IDirectDrawSurface2::Blt**, **IDirectDrawSurface2::BltBatch**, and **IDirectDrawSurface2::UpdateOverlay** methods on surfaces to which the parent DirectDrawClipper object is attached.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_CLIPPERISUSINGHWND

  DDERR_INVALIDCLIPLIST

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_OUTOFMEMORY

lpClipList
Either an address to a valid **RGNDATA** structure or NULL. If there is an existing clip list associated with the DirectDrawClipper object and this value is NULL, the clip list will be deleted.

dwFlags
This parameter is currently not used and must be set to 0.

The clip list cannot be set if a window handle is already associated with the DirectDrawClipper object. Note that the **IDirectDrawSurface2::BltFast** method cannot clip.

The **RGNDATA** structure used with this method has the following syntax.

```
typedef struct _RGNDATA {
    RGNDATAHEADER rdh;
    char          Buffer[1];
```

```
} RGNDATA;
```

The **rdh** member of the **RGNDATA** structure is an **RGNDATAHEADER** structure that has the following syntax.

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize;
    DWORD iType;
    DWORD nCount;
    DWORD nRgnSize;
    RECT  rcBound;
} RGNDATAHEADER;
```

For more information about these structures, see the documentation in the Win32 Software Development Kit.

See also **IDirectDrawClipper::GetClipList**, **IDirectDrawSurface2::Blt**, **IDirectDrawSurface2::BltFast**, **IDirectDrawSurface2::BltBatch**, **IDirectDrawSurface2::UpdateOverlay**

IDirectDrawClipper::SetHWnd

```
HRESULT SetHWnd(DWORD dwFlags, HWND hWnd);
```

Sets the window handle that will obtain the clipping information.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDCLIPLIST

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_OUTOFMEMORY

dwFlags
This parameter is currently not used and must be set to 0.

hWnd
Window handle that obtains the clipping information.

See also **IDirectDrawClipper::GetHWnd**

IDirectDrawPalette

Applications use the methods of the **IDirectDrawPalette** interface to create DirectDrawPalette objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see *DirectDrawPalette Objects*.

The methods of the **IDirectDrawPalette** interface can be organized into the following groups:

Allocating memory	**Initialize**
Palette capabilities	**GetCaps**
Palette entries	**GetEntries**
	SetEntries

The **IDirectDrawPalette** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

IDirectDrawPalette::GetCaps

```
HRESULT GetCaps(LPDWORD lpdwCaps);
```

Retrieves the capabilities of this palette object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lpdwCaps

Flag from the **dwPalCaps** member of the **DDCAPS** structure that defines palette capabilities:

DDPCAPS_4BIT

DDPCAPS_8BIT

DDPCAPS_8BITENTRIES

DDPCAPS_ALLOW256

DDPCAPS_PRIMARYSURFACE

DDPCAPS_PRIMARYSURFACELEFT

DDPCAPS_VSYNC

IDirectDrawPalette::GetEntries

```
HRESULT GetEntries(DWORD dwFlags, DWORD dwBase,
    DWORD dwNumEntries, LPPALETTEENTRY lpEntries);
```

Queries palette values from a DirectDrawPalette object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOTPALETTIZED

dwFlags
This parameter is currently not used and must be set to 0.

dwBase
Start of the entries that should be retrieved sequentially.

dwNumEntries
Number of palette entries that can fit in the address specified in *lpEntries*. The colors of each palette entry are returned in sequence, from the value of the *dwStartingEntry* parameter through the value of the *dwCount* parameter minus 1. (These parameters are set by **IDirectDrawPalette::SetEntries**.)

lpEntries
Address of the palette entries. The palette entries are 1 byte each if the DDPCAPS_8BITENTRIES flag is set and 4 bytes otherwise. Each field is a color description.

See also **IDirectDrawPalette::SetEntries**

IDirectDrawPalette::Initialize

```
HRESULT Initialize(LPDIRECTDRAW lpDD, DWORD dwFlags,
    LPPALETTEENTRY lpDDColorTable);
```

Initializes the DirectDrawPalette object.

- Returns **DDERR_ALREADYINITIALIZED**.

lpDD
Address of the DirectDraw structure that represents the DirectDraw object.

dwFlags and *lpDDColorTable*
These parameters are currently not used and must be set to 0.

This method is provided for compliance with the Component Object Model (COM) protocol. Because the DirectDrawPalette object is initialized when it is created, this method always returns DDERR_ALREADYINITIALIZED.

See also **IUnknown::AddRef**, **IUnknown::QueryInterface**, **IUnknown::Release**

IDirectDrawPalette::SetEntries

```
HRESULT SetEntries(DWORD dwFlags, DWORD dwStartingEntry,
    DWORD dwCount, LPPALETTEENTRY lpEntries);
```

Changes entries in a DirectDrawPalette object immediately.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOPALETTEATTACHED

  DDERR_NOTPALETTIZED

  DDERR_UNSUPPORTED

dwFlags
This parameter is currently not used and must be set to 0.

dwStartingEntry
First entry to be set.

dwCount
Number of palette entries to be changed.

lpEntries
Address of the palette entries. The palette entries are 1 byte each if the DDPCAPS_8BITENTRIES flag is set and 4 bytes otherwise. Each field is a color description.

The palette must have been attached to a surface by using the **IDirectDrawSurface2::SetPalette** method before **IDirectDrawPalette::SetEntries** can be used.

See also **IDirectDrawPalette::GetEntries**, **IDirectDrawSurface2::SetPalette**

IDirectDrawSurface2

Applications use the methods of the **IDirectDrawSurface2** interface to create DirectDrawSurface objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see *DirectDrawSurface Objects*.

The methods of the **IDirectDrawSurface2** interface can be organized into the following groups:

Allocating memory	**Initialize**

	IsLost
	Restore
Attaching surfaces	**AddAttachedSurface**
	DeleteAttachedSurface
	EnumAttachedSurfaces
	GetAttachedSurface
Blitting	**Blt**
	BltBatch
	BltFast
Color keys	**GetColorKey**
	SetColorKey
Device contexts	**GetDC**
	ReleaseDC
Flipping surfaces	**Flip**
Locking surfaces	**Lock**
	PageLock
	PageUnlock
	Unlock
Miscellaneous	**GetDDInterface**
Overlays	**AddOverlayDirtyRect**
	EnumOverlayZOrders
	GetOverlayPosition
	SetOverlayPosition
	UpdateOverlay
	UpdateOverlayDisplay
	UpdateOverlayZOrder
Status	**GetBltStatus**

	GetFlipStatus
Surface capabilities	**GetCaps**
Surface clipper	**GetClipper** **SetClipper**
Surface description	**GetPixelFormat** **GetSurfaceDesc**
Surface palettes	**GetPalette** **SetPalette**

The **IDirectDrawSurface2** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The **IUnknown** interface supports the following three methods:

AddRef

QueryInterface

Release

IDirectDrawSurface2::AddAttachedSurface

```
HRESULT AddAttachedSurface(
    LPDIRECTDRAWSURFACE2 lpDDSAttachedSurface);
```

Attaches a surface to another surface.

- Returns DD_OK if successful, or one of the following error values otherwise:
  DDERR_CANNOTATTACHSURFACE
  DDERR_GENERIC
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS
  DDERR_SURFACEALREADYATTACHED
  DDERR_SURFACELOST
  DDERR_WASSTILLDRAWING

lpDDSAttachedSurface
Address of the DirectDraw surface that is to be attached.

Possible attachments include z-buffers, alpha channels, and back buffers. Some attachments automatically break other attachments. For example, the 3D z-buffer can be attached only to one back buffer at a time. Attachment is not bidirectional, and a surface cannot be attached to itself. Emulated surfaces (in system memory) cannot be attached to non-emulated surfaces. Unless one surface is a texture map, the two attached surfaces must be the same size. A flipping surface cannot be attached to another flipping surface of the same type; however, attaching two surfaces of different types is allowed. For example, a flipping z-buffer can be attached to a regular flipping surface. If a non-flipping surface is attached to another non-flipping surface of the same type, the two surfaces will become a flipping chain. If a non-flipping surface is attached to a flipping surface, it becomes part of the existing flipping chain. Additional surfaces can be added to this chain, and each call of the **IDirectDrawSurface2::Flip** method will advance one step through the surfaces.

See also **IDirectDrawSurface2::DeleteAttachedSurface**, **IDirectDrawSurface2::EnumAttachedSurfaces**, **IDirectDrawSurface2::Flip**

IDirectDrawSurface2::AddOverlayDirtyRect

```
HRESULT AddOverlayDirtyRect(LPRECT lpRect);
```

Builds the list of the rectangles that need to be updated the next time the **IDirectDrawSurface2::UpdateOverlayDisplay** method is called.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

  DDERR_UNSUPPORTED

lpRect
Address of the **RECT** structure that needs to be updated.

This method is used for the software implementation. It is not needed if the overlay support is provided by the hardware.

See also **IDirectDrawSurface2::UpdateOverlayDisplay**

IDirectDrawSurface2::Blt

```
HRESULT Blt(LPRECT lpDestRect, LPDIRECTDRAWSURFACE2 lpDDSrcSurface,
    LPRECT lpSrcRect, DWORD dwFlags, LPDDBLTFX lpDDBltFx);
```

Performs a bit block transfer.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDCLIPLIST

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDRECT

  DDERR_NOALPHAHW

  DDERR_NOBLTHW

  DDERR_NOCLIPLIST

  DDERR_NODDROPSHW

  DDERR_NOMIRRORHW

  DDERR_NORASTEROPHW

  DDERR_NOROTATIONHW

  DDERR_NOSTRETCHHW

  DDERR_NOZBUFFERHW

  DDERR_SURFACEBUSY

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

lpDestRect

Address of a **RECT** structure that defines the upper-left and lower-right points of the rectangle on the destination surface to be blitted to.

lpDDSrcSurface

Address of the DirectDraw surface that is the source for the blit operation.

lpSrcRect

Address of a **RECT** structure that defines the upper-left and lower-right points of the rectangle on the source surface to be blitted from.

dwFlags

DDBLT_ALPHADEST

Uses either the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for this blit.

DDBLT_ALPHADESTCONSTOVERRIDE

Uses the **dwAlphaDestConst** member of the **DDBLTFX** structure as the alpha channel for the destination surface for this blit.

DDBLT_ALPHADESTNEG

Indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHADESTSURFACEOVERRIDE

Uses the **lpDDSAlphaDest** member of the **DDBLTFX** structure as the alpha channel for the destination for this blit.

DDBLT_ALPHAEDGEBLEND

Uses the **dwAlphaEdgeBlend** member of the **DDBLTFX** structure as the alpha channel for the edges of the image that border the color key colors.

DDBLT_ALPHASRC

Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the alpha channel for this blit.

DDBLT_ALPHASRCCONSTOVERRIDE

Uses the **dwAlphaSrcConst** member of the **DDBLTFX** structure as the alpha channel for the source for this blit.

DDBLT_ALPHASRCNEG

Indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHASRCSURFACEOVERRIDE

Uses the **lpDDSAlphaSrc** member of the **DDBLTFX** structure as the alpha channel for the source for this blit.

DDBLT_ASYNC

Performs this blit asynchronously through the FIFO in the order received. If no room is available in the FIFO hardware, the call fails.

DDBLT_COLORFILL

Uses the **dwFillColor** member of the **DDBLTFX** structure as the RGB color that fills the destination rectangle on the destination surface.

DDBLT_DDFX

Uses the **dwDDFX** member of the **DDBLTFX** structure to specify the effects to use for this blit.

DDBLT_DDROPS

Uses the **dwDDROPS** member of the **DDBLTFX** structure to specify the raster operations (ROPS) that are not part of the Win32 API.

DDBLT_DEPTHFILL

Uses the **dwFillDepth** member of the **DDBLTFX** structure as the depth value with which to fill the destination rectangle on the destination z-buffer surface.

DDBLT_KEYDEST

Uses the color key associated with the destination surface.

DDBLT_KEYDESTOVERRIDE

Uses the **dckDestColorkey** member of the **DDBLTFX** structure as the color key for the destination surface.

DDBLT_KEYSRC

Uses the color key associated with the source surface.

DDBLT_KEYSRCOVERRIDE

Uses the **dckSrcColorkey** member of the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP

Uses the **dwROP** member of the **DDBLTFX** structure for the ROP for this blit. These ROPs are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE

Uses the **dwRotationAngle** member of the **DDBLTFX** structure as the rotation angle (specified in 1/100th of a degree) for the surface.

DDBLT_WAIT

Postpones the DDERR_WASSTILLDRAWING return value if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

DDBLT_ZBUFFER

Performs a z-buffered blit using the z-buffers attached to the source and destination surfaces and the **dwZBufferOpCode** member of the **DDBLTFX** structure as the z-buffer opcode.

DDBLT_ZBUFFERDESTCONSTOVERRIDE

Performs a z-buffered blit using the **dwZDestConst** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the destination.

DDBLT_ZBUFFERDESTOVERRIDE

Performs a z-buffered blit using the **lpDDSZBufferDest** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the destination.

DDBLT_ZBUFFERSRCCONSTOVERRIDE

Performs a z-buffered blit using the **dwZSrcConst** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the source.

DDBLT_ZBUFFERSRCOVERRIDE

Performs a z-buffered blit using the **lpDDSZBufferSrc** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the source.

lpDDBltFx

Address of the **DDBLTFX** structure.

This method is capable of synchronous or asynchronous blits, either display memory to display memory, display memory to system memory, system memory to display memory, or system memory to system memory. The blits can be performed by using z-information, alpha information, source color keys, and destination color keys. Arbitrary stretching or shrinking will be performed if the source and destination rectangles are not the same size.

Typically, **IDirectDrawSurface2::Blt** returns immediately with an error if the blitter is busy and the blit could not be set up. The DDBLT_WAIT flag can alter this behavior so that the method will either wait until the blit can be set up or another error occurs before it returns.

IDirectDrawSurface2::BltBatch

```
HRESULT BltBatch(LPDDBLTBATCH lpDDBltBatch,
    DWORD dwCount, DWORD dwFlags);
```

Performs a sequence of **IDirectDrawSurface2::Blt** operations from several sources to a single destination. This method is currently only a stub; it has not yet been implemented.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC
  DDERR_INVALIDCLIPLIST
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS
  DDERR_INVALIDRECT
  DDERR_NOALPHAHW
  DDERR_NOBLTHW
  DDERR_NOCLIPLIST
  DDERR_NODDROPSHW
  DDERR_NOMIRRORHW
  DDERR_NORASTEROPHW
  DDERR_NOROTATIONHW
  DDERR_NOSTRETCHHW
  DDERR_NOZBUFFERHW
  DDERR_SURFACEBUSY
  DDERR_SURFACELOST
  DDERR_UNSUPPORTED

lpDDBltBatch
Address of the first **DDBLTBATCH** structure that defines the parameters for the blit operations.

dwCount
Number of blit operations to be performed.

dwFlags
This parameter is currently not used and must be set to 0.

IDirectDrawSurface2::BltFast

```
HRESULT BltFast(DWORD dwX, DWORD dwY,
    LPDIRECTDRAWSURFACE2 lpDDSrcSurface, LPRECT lpSrcRect,
    DWORD dwTrans);
```

Performs a source copy blit or transparent blit by using a source color key or destination color key. This method always attempts an asynchronous blit if it is supported by the hardware.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_EXCEPTION

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDRECT

  DDERR_NOBLTHW

  DDERR_SURFACEBUSY

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

dwX and *dwY*
The x- and y-coordinates to blit to on the destination surface.

lpDDSrcSurface
Address of the DirectDraw surface that is the source for the blit operation.

lpSrcRect
Address of a **RECT** structure that defines the upper-left and lower-right points of the rectangle on the source surface to be blitted from.

dwTrans
Type of transfer.

DDBLTFAST_DESTCOLORKEY

Specifies a transparent blit that uses the destination's color key.

DDBLTFAST_NOCOLORKEY

Specifies a normal copy blit with no transparency.

DDBLTFAST_SRCCOLORKEY

Specifies a transparent blit that uses the source's color key.

DDBLTFAST_WAIT

Postpones the DDERR_WASSTILLDRAWING message if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

This method works only on display memory surfaces and cannot clip when blitting. The software implementation of **IDirectDrawSurface2::BltFast** is 10 percent faster than the **IDirectDrawSurface2::Blt** method. However, there is no speed difference between the two if display hardware is being used.

Typically, **IDirectDrawSurface2::BltFast** returns immediately with an error if the blitter is busy and the blit cannot be set up. You can use the DDBLTFAST_WAIT flag, however, if you want this method to not return until either the blit can be set up or another error occurs.

IDirectDrawSurface2::DeleteAttachedSurface

```
HRESULT DeleteAttachedSurface(DWORD dwFlags,
    LPDIRECTDRAWSURFACE2 lpDDSAttachedSurface);
```

Detaches two attached surfaces. The detached surface is not released.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_CANNOTDETACHSURFACE

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_SURFACELOST

  DDERR_SURFACENOTATTACHED

dwFlags
This parameter is currently not used and must be set to 0.

lpDDSAttachedSurface
Address of the DirectDraw surface to be detached. If this parameter is NULL, all attached surfaces are detached.

Implicit attachments, those formed by DirectDraw rather than the **IDirectDrawSurface2::AddAttachedSurface** method, cannot be detached. Detaching surfaces from a flipping chain can alter other surfaces in the chain. If a front buffer is detached from a flipping chain, the next surface in the chain becomes the front buffer, and the following surface becomes the back buffer. If a back buffer is detached from a chain, the following surface becomes a back buffer. If a plain surface is detached from a chain, the chain simply becomes shorter. If a flipping chain has only two surfaces and they are detached, the chain is destroyed and both surfaces return to their previous designations.

See also **IDirectDrawSurface2::Flip**

IDirectDrawSurface2::EnumAttachedSurfaces

```
HRESULT EnumAttachedSurfaces(LPVOID lpContext,
```

```
    LPDDENUMSURFACESCALLBACK lpEnumSurfacesCallback);
```

Enumerates all the surfaces attached to a given surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_SURFACELOST

lpContext
Address of the application-defined structure that is passed to the enumeration member every time it is called.

lpEnumSurfacesCallback
Address of the **EnumSurfacesCallback** function that will be called for each surface that is attached to this surface.

IDirectDrawSurface2::EnumOverlayZOrders

```
HRESULT EnumOverlayZOrders(DWORD dwFlags, LPVOID lpContext,
    LPDDENUMSURFACESCALLBACK lpfnCallback);
```

Enumerates the overlay surfaces on the specified destination. The overlays can be enumerated in front-to-back or back-to-front order.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

dwFlags
One of the following flags:

DDENUMOVERLAYZ_BACKTOFRONT
Enumerates overlays back to front.

DDENUMOVERLAYZ_FRONTTOBACK
Enumerates overlays front to back.

lpContext
Address of the user-defined context that will be passed to the callback function for each overlay surface.

lpfnCallback
Address of the **fnCallback** function that will be called for each surface being overlaid on this surface.

IDirectDrawSurface2::Flip

```
HRESULT Flip(
    LPDIRECTDRAWSURFACE2 lpDDSurfaceTargetOverride,
    DWORD dwFlags);
```

Makes the surface memory associated with the DDSCAPS_BACKBUFFER surface become associated with the front-buffer surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOFLIPHW

  DDERR_NOTFLIPPABLE

  DDERR_SURFACEBUSY

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

  DDERR_WASSTILLDRAWING

lpDDSurfaceTargetOverride

Address of the DirectDraw surface that will be flipped to. The default for this parameter is NULL, in which case **IDirectDrawSurface2::Flip** cycles through the buffers in the order they are attached to each other. This parameter is used only as an override.

dwFlags

DDFLIP_WAIT

Typically, if the flip cannot be set up because the state of the display hardware is not appropriate, the DDERR_WASSTILLDRAWING error returns immediately and no flip occurs. Setting this flag causes **IDirectDrawSurface2::Flip** to continue trying to flip if it receives the DDERR_WASSTILLDRAWING error from the HAL. **IDirectDrawSurface2::Flip** does not return until the flipping operation has been successfully set up, or if another error, such as DDERR_SURFACEBUSY, is returned.

This method can be called only by a surface that has the DDSCAPS_FLIP and DDSCAPS_FRONTBUFFER values set. The display memory previously associated with the front buffer is associated with the back buffer. If there is more than one back buffer, a ring is formed and the surface memory buffers cycle one step through it every time **IDirectDrawSurface2::Flip** is called.

The *lpDDSurfaceTargetOverride* parameter is used in rare cases when the back buffer is not the buffer that should become the front buffer. Typically this parameter is NULL.

The **IDirectDrawSurface2::Flip** method will always be synchronized with the vertical blank.

See also **IDirectDrawSurface2::GetFlipStatus**

IDirectDrawSurface2::GetAttachedSurface

```
HRESULT GetAttachedSurface(LPDDSCAPS lpDDSCaps,
    LPDIRECTDRAWSURFACE2 FAR * lplpDDAttachedSurface);
```

Obtains the attached surface that has the specified capabilities.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOTFOUND

  DDERR_SURFACELOST

lpDDSCaps
Address of a **DDSCAPS** structure that contains the hardware capabilities of the surface.

lplpDDAttachedSurface
Address of a pointer to a DirectDraw surface that will be attached to the current DirectDraw surface specified by the *lpDDSurface* parameter in the **EnumSurfacesCallback** callback function, and that has capabilities that match those specified by the *lpDDSCaps* parameter.

Attachments are used to connect multiple DirectDrawSurface objects into complex structures, like the ones needed to support 3D page flipping with z-buffers. This method fails if more than one surface is attached that matches the capabilities requested. In this case, the application must use the **IDirectDrawSurface2::EnumAttachedSurfaces** method to obtain the attached surfaces.

IDirectDrawSurface2::GetBltStatus

```
HRESULT GetBltStatus(DWORD dwFlags);
```

Obtains the blitter status.

- Returns DD_OK if a blitter is present, DDERR_WASSTILLDRAWING if the blitter is busy, DDERR_NOBLTHW if there is no blitter, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

DDERR_NOBLTHW

DDERR_SURFACEBUSY

DDERR_SURFACELOST

DDERR_UNSUPPORTED

DDERR_WASSTILLDRAWING

dwFlags

One of the following flags:

DDGBS_CANBLT

Inquires whether a blit involving this surface can occur immediately, and returns DD_OK if the blit can be completed.

DDGBS_ISBLTDONE

Inquires whether the blit is done, and returns DD_OK if the last blit on this surface has completed.

IDirectDrawSurface2::GetCaps

```
HRESULT GetCaps(LPDDSCAPS lpDDSCaps);
```

Retrieves the capabilities of the surface. These capabilities are not necessarily related to the capabilities of the display device.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lpDDSCaps

Address of a **DDSCAPS** structure that will be filled with the hardware capabilities of the surface.

IDirectDrawSurface2::GetClipper

```
HRESULT GetClipper(LPDIRECTDRAWCLIPPER FAR * lplpDDClipper);
```

Retrieves the DirectDrawClipper object associated with this surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOCLIPPERATTACHED

lplpDDClipper
Address of a pointer to the DirectDrawClipper object associated with the surface.

See also **IDirectDrawSurface2::SetClipper**

IDirectDrawSurface2::GetColorKey

```
HRESULT GetColorKey(DWORD dwFlags, LPDDCOLORKEY lpDDColorKey);
```

Retrieves the color key value for the DirectDrawSurface object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOCOLORKEY

  DDERR_NOCOLORKEYHW

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

dwFlags
Determines which color key is requested.

DDCKEY_DESTBLT

Set if the structure specifies a color key or color space to be used as a destination color key for blit operations.

DDCKEY_DESTOVERLAY

Set if the structure specifies a color key or color space to be used as a destination color key for overlay operations.

DDCKEY_SRCBLT

Set if the structure specifies a color key or color space to be used as a source color key for blit operations.

DDCKEY_SRCOVERLAY

Set if the structure specifies a color key or color space to be used as a source color key for overlay operations.

lpDDColorKey
Address of the **DDCOLORKEY** structure that will be filled with the current values for the specified color key of the DirectDrawSurface object.

See also **IDirectDrawSurface2::SetColorKey**

IDirectDrawSurface2::GetDC

```
HRESULT GetDC(HDC FAR * lphDC);
```

Creates a GDI-compatible handle of a device context for the surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_DCALREADYCREATED

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

  DDERR_WASSTILLDRAWING

lphDC
Address for the returned handle of a device context.

This method uses an internal version of the **IDirectDrawSurface2::Lock** method to lock the surface. The surface remains locked until the **IDirectDrawSurface2::ReleaseDC** method is called.

See also **IDirectDrawSurface2::Lock**

IDirectDrawSurface2::GetDDInterface

```
HRESULT GetDDInterface(LPVOID FAR *lplpDD);
```

Retrieves an interface to the DirectDraw object that was used to create the surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lplpDD
Address of a pointer that will be filled with a valid DirectDraw pointer if the call succeeds.

This method was not implemented in the **IDirectDraw** interface.

IDirectDrawSurface2::GetFlipStatus

```
HRESULT GetFlipStatus(DWORD dwFlags);
```

Indicates whether the surface has finished its flipping process.

- Returns DD_OK if successful, **DDERR_WASSTILLDRAWING** if the surface has not finished its flipping process, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

  DDERR_SURFACEBUSY

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

dwFlags
One of the following flags:

DDGFS_CANFLIP

Inquires whether this surface can be flipped immediately and returns DD_OK if the flip can be completed.

DDGFS_ISFLIPDONE

Inquires whether the flip has finished and returns DD_OK if the last flip on this surface has completed.

See also **IDirectDrawSurface2::Flip**

IDirectDrawSurface2::GetOverlayPosition

```
HRESULT GetOverlayPosition(LPLONG lplX, LPLONG lplY);
```

Given a visible, active overlay surface (DDSCAPS_OVERLAY flag set), this method returns the display coordinates of the surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDPOSITION

  DDERR_NOOVERLAYDEST

  DDERR_NOTAOVERLAYSURFACE

  DDERR_OVERLAYNOTVISIBLE

  DDERR_SURFACELOST

lplX and *lplY*
Addresses of the x- and y-display coordinates.

See also **IDirectDrawSurface2::SetOverlayPosition**, **IDirectDrawSurface2::UpdateOverlay**

IDirectDrawSurface2::GetPalette

```
HRESULT GetPalette(LPDIRECTDRAWPALETTE FAR * lplpDDPalette);
```

Retrieves the DirectDrawPalette structure associated with this surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOEXCLUSIVEMODE

  DDERR_NOPALETTEATTACHED

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

lplpDDPalette
Address of a pointer to a DirectDrawPalette structure associated with this surface. This parameter will be set to NULL if no DirectDrawPalette structure is associated with this surface. If the surface is the primary surface, or a back buffer to the primary surface, and the primary surface is in 8-bpp mode, this parameter will contain a pointer to the system palette.

See also **IDirectDrawSurface2::SetPalette**

IDirectDrawSurface2::GetPixelFormat

```
HRESULT GetPixelFormat(LPDDPIXELFORMAT lpDDPixelFormat);
```

Retrieves the color and pixel format of the surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

lpDDPixelFormat
Address of the **DDPIXELFORMAT** structure that will be filled with a detailed description of the current pixel and color space format of the surface.

IDirectDrawSurface2::GetSurfaceDesc

```
HRESULT GetSurfaceDesc(LPDDSURFACEDESC lpDDSurfaceDesc);
```

Retrieves a **DDSURFACEDESC** structure that describes the surface in its current condition.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

lpDDSurfaceDesc
Address of a **DDSURFACEDESC** structure that will be filled with the current description of this surface.

See also **DDSURFACEDESC**

IDirectDrawSurface2::Initialize

```
HRESULT Initialize(LPDIRECTDRAW lpDD,
    LPDDSURFACEDESC lpDDSurfaceDesc);
```

Initializes a DirectDrawSurface object.

- Returns **DDERR_ALREADYINITIALIZED**.

lpDD
Address of the DirectDraw structure that represents the DirectDraw object.

lpDDSurfaceDesc
Address of a **DDSURFACEDESC** structure that will be filled with the relevant details about the surface.

This method is provided for compliance with the Component Object Model (COM) protocol. Because the DirectDrawSurface object is initialized when it is created, this method always returns DDERR_ALREADYINITIALIZED.

See also **IUnknown::AddRef**, **IUnknown::QueryInterface**, **IUnknown::Release**

IDirectDrawSurface2::IsLost

```
HRESULT IsLost();
```

Determines if the surface memory associated with a DirectDrawSurface object has been freed.

- Returns DD_OK if the memory has not been freed, or one of the following error values otherwise:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_SURFACELOST

You can use this method to reallocate surface memory. When a DirectDrawSurface object loses its surface memory, most methods return DDERR_SURFACELOST and perform no other action.

Surfaces can lose their memory when the mode of the display card is changed, or when an application receives exclusive access to the display card and frees all of the surface memory currently allocated on the display card.

See also **IDirectDrawSurface2::Restore**

IDirectDrawSurface2::Lock

```
HRESULT Lock(LPRECT lpDestRect, LPDDSURFACEDESC lpDDSurfaceDesc,
    DWORD dwFlags, HANDLE hEvent);
```

Obtains a pointer to the surface memory.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_OUTOFMEMORY

  DDERR_SURFACEBUSY

  DDERR_SURFACELOST

  DDERR_WASSTILLDRAWING

lpDestRect

Address of a **RECT** structure that identifies the region of surface that is being locked.

lpDDSurfaceDesc

Address of a **DDSURFACEDESC** structure that will be filled with the relevant details about the surface.

dwFlags

DDLOCK_EVENT

Triggers the event when **IDirectDrawSurface2::Lock** can return the surface memory pointer requested. This flag is set if an event handle is being passed to **IDirectDrawSurface2::Lock**. If multiple locks of this type are placed on a surface, events are triggered in FIFO order.

DDLOCK_READONLY

Indicates that the surface being locked will only be read from.

DDLOCK_SURFACEMEMORYPTR

Indicates that a valid memory pointer to the top of the specified rectangle should be returned. If no rectangle is specified, a pointer to the top of the surface is returned. This is the default.

DDLOCK_WAIT

Typically, if a lock cannot be obtained because a blit operation is in progress, a DDERR_WASSTILLDRAWING error will be returned immediately. If this flag is set, however, **IDirectDrawSurface2::Lock** retries until a lock is obtained or another error, such as DDERR_SURFACEBUSY, occurs.

DDLOCK_WRITEONLY

Indicates that the surface being locked will only be written to.

hEvent

Handle of a system event that is triggered when the surface is ready to be locked.

After the pointer is obtained, your application can access the surface memory until the corresponding **IDirectDrawSurface2::Unlock** method is called. After this method is called, the pointer to the surface memory is no longer valid.

Your application cannot blit from a region of a surface that is locked. If a blit is attempted on a locked surface, the blit returns either a DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES error value.

Typically, **IDirectDrawSurface2::Lock** returns immediately with an error when a lock cannot be obtained because a blit is in progress. You can use the DDLOCK_WAIT flag if you want the method to continue trying to obtain a lock.

To prevent display memory from being lost during access to a surface, DirectDraw holds the Win16 lock between **IDirectDrawSurface2::Lock** and **IDirectDrawSurface2::Unlock** operations. The Win16 lock is the critical section that serializes access to GDI and USER. Although this technique allows direct access to display memory and prevents other applications from changing the mode during this access, it stops Windows from running, so **IDirectDrawSurface2::Lock**/**IDirectDrawSurface2::Unlock** and **IDirectDrawSurface2::GetDC**/**IDirectDrawSurface2::ReleaseDC** periods should be kept short. Unfortunately, because Windows is stopped, GUI debuggers cannot be used between **IDirectDrawSurface2::Lock**/**IDirectDrawSurface2::Unlock** or **IDirectDrawSurface2::GetDC**/**IDirectDrawSurface2::ReleaseDC** operations.

See also **IDirectDrawSurface2::Unlock**, **IDirectDrawSurface2::GetDC**, **IDirectDrawSurface2::ReleaseDC**

IDirectDrawSurface2::PageLock

```
HRESULT PageLock(DWORD dwFlags);
```

Prevents a system-memory surface from being paged out while a blit operation using direct memory access (DMA) transfers to or from system memory is in progress.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_CANTPAGELOCK

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_SURFACELOST

dwFlags
This parameter is currently not used and must be set to 0.

The performance of the operating system could be negatively affected if too much memory is locked.

A lock count is maintained for each surface and is incremented each time **IDirectDrawSurface2::PageLock** is called for that surface. The count is decremented when **IDirectDrawSurface2::PageUnlock** is called. When the count reaches 0, the memory is unlocked and can then be paged by the operating system.

This method works only on system-memory surfaces; it will not page lock a display-memory surface or an emulated primary surface. If an application calls this method on a display memory surface, the method will do nothing except return DD_OK.

This method was not implemented in the **IDirectDraw** interface.

See also **IDirectDrawSurface2::PageUnlock**

IDirectDrawSurface2::PageUnlock

```
HRESULT PageUnlock(DWORD dwFlags);
```

Unlocks a system-memory surface, allowing it to be paged out.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_CANTPAGEUNLOCK

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOTPAGELOCKED

DDERR_SURFACELOST

dwFlags
This parameter is currently not used and must be set to 0.

A lock count is maintained for each surface and is incremented each time **IDirectDrawSurface2::PageLock** is called for that surface. The count is decremented when **IDirectDrawSurface2::PageUnlock** is called. When the count reaches 0, the memory is unlocked and can then be paged by the operating system.

This method works only on system-memory surfaces; it will not page unlock a display-memory surface or an emulated primary surface. If an application calls this method on a display-memory surface, this method will do nothing except return DD_OK.

This method was not implemented in the **IDirectDraw** interface.

See also **IDirectDrawSurface2::PageLock**

IDirectDrawSurface2::ReleaseDC

```
HRESULT ReleaseDC(HDC hDC);
```

Releases the handle of a device context previously obtained by using the **IDirectDrawSurface2::GetDC** method.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

hDC
Handle of a device context previously obtained by **IDirectDrawSurface2::GetDC**.

This method also unlocks the surface previously locked when the **IDirectDrawSurface2::GetDC** method was called.

See also **IDirectDrawSurface2::GetDC**

IDirectDrawSurface2::Restore

```
HRESULT Restore();
```

Restores a surface that has been lost. This occurs when the surface memory associated with the DirectDrawSurface object has been freed.

- Returns DD_OK is successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_IMPLICITLYCREATED

  DDERR_INCOMPATIBLEPRIMARY

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOEXCLUSIVEMODE

  DDERR_OUTOFMEMORY

  DDERR_UNSUPPORTED

  DDERR_WRONGMODE

Surfaces can be lost because the mode of the display card was changed or because an application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. When a DirectDrawSurface object loses its surface memory, many methods will return DDERR_SURFACELOST and perform no other function. The **IDirectDrawSurface2::Restore** method will reallocate surface memory and reattach it to the DirectDrawSurface object.

A single call to this method will restore a DirectDrawSurface object's associated implicit surfaces (back buffers, and so on). An attempt to restore an implicitly created surface will result in an error. **IDirectDrawSurface2::Restore** will not work across explicit attachments created by using the **IDirectDrawSurface2::AddAttachedSurface** method — each of these surfaces must be restored individually.

See also **IDirectDrawSurface2::IsLost**, **IDirectDrawSurface2::AddAttachedSurface**

IDirectDrawSurface2::SetClipper

```
HRESULT SetClipper(LPDIRECTDRAWCLIPPER lpDDClipper);
```

Attaches a DirectDrawClipper object to a DirectDrawSurface object.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

DDERR_NOCLIPPERATTACHED

lpDDClipper
Address of the DirectDrawClipper structure representing the DirectDrawClipper object that will be attached to the DirectDrawSurface object. If this parameter is NULL, the current DirectDrawClipper object will be detached.

This method is primarily used by surfaces that are being overlaid on or blitted to the primary surface. However, it can be used on any surface. After a DirectDrawClipper object has been attached and a clip list is associated with it, the DirectDrawClipper object will be used for the **IDirectDrawSurface2::Blt**, **IDirectDrawSurface2::BltBatch**, and **IDirectDrawSurface2::UpdateOverlay** operations involving the parent DirectDrawSurface object. This method can also detach a DirectDrawSurface object's current DirectDrawClipper object.

If this method 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.

See also **IDirectDrawSurface2::GetClipper**

IDirectDrawSurface2::SetColorKey

```
HRESULT SetColorKey(DWORD dwFlags, LPDDCOLORKEY lpDDColorKey);
```

Sets the color key value for the DirectDrawSurface object if the hardware supports color keys on a per surface basis.

- Returns DD_OK is successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

  DDERR_NOOVERLAYHW

  DDERR_NOTAOVERLAYSURFACE

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

  DDERR_WASSTILLDRAWING

dwFlags
Determines which color key is requested.

DDCKEY_COLORSPACE
Set if the structure contains a color space. Not set if the structure contains

a single color key.

DDCKEY_DESTBLT

Set if the structure specifies a color key or color space to be used as a destination color key for blit operations.

DDCKEY_DESTOVERLAY

Set if the structure specifies a color key or color space to be used as a destination color key for overlay operations.

DDCKEY_SRCBLT

Set if the structure specifies a color key or color space to be used as a source color key for blit operations.

DDCKEY_SRCOVERLAY

Set if the structure specifies a color key or color space to be used as a source color key for overlay operations.

lpDDColorKey

Address of the **DDCOLORKEY** structure that contains the new color key values for the DirectDrawSurface object.

See also **IDirectDrawSurface2::GetColorKey**

IDirectDrawSurface2::SetOverlayPosition

```
HRESULT SetOverlayPosition(LONG lX, LONG lY);
```

Changes the display coordinates of an overlay surface.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

lX and *lY*

New x- and y-display coordinates.

See also **IDirectDrawSurface2::GetOverlayPosition**, **IDirectDrawSurface2::UpdateOverlay**

IDirectDrawSurface2::SetPalette

```
HRESULT SetPalette(LPDIRECTDRAWPALETTE lpDDPalette);
```

Attaches the specified DirectDrawPalette object to a surface. The surface uses this palette for all subsequent operations. The palette change takes place immediately, without regard to refresh timing.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

  DDERR_NOEXCLUSIVEMODE

  DDERR_NOPALETTEATTACHED

  DDERR_NOPALETTEHW

  DDERR_NOT8BITCOLOR

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

lpDDPalette
: Address of the DirectDrawPalette structure that this surface should use for future operations.

If this method is called several times consecutively on the same surface for the same palette, the reference count for the palette is incremented only once. Subsequent calls do not affect the palette's reference count.

See also **IDirectDrawSurface2::GetPalette**, **IDirectDraw2::CreatePalette**

IDirectDrawSurface2::Unlock

```
HRESULT Unlock(LPVOID lpSurfaceData);
```

Notifies DirectDraw that the direct surface manipulations are complete.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDRECT

  DDERR_NOTLOCKED

  DDERR_SURFACELOST

lpSurfaceData
Address of a pointer returned by the **IDirectDrawSurface2::Lock** method. Because it is possible to call **IDirectDrawSurface2::Lock** multiple times for the same surface with different destination rectangles, this pointer links the calls to the **IDirectDrawSurface2::Lock** and **IDirectDrawSurface2::Unlock** methods.

See also **IDirectDrawSurface2::Lock**

IDirectDrawSurface2::UpdateOverlay

```
HRESULT UpdateOverlay(LPRECT lpSrcRect,
    LPDIRECTDRAWSURFACE2 lpDDDestSurface,
    LPRECT lpDestRect, DWORD dwFlags,
    LPDDOVERLAYFX lpDDOverlayFx);
```

Repositions or modifies the visual attributes of an overlay surface. These surfaces must have the DDSCAPS_OVERLAY value set.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_GENERIC

  DDERR_HEIGHTALIGN

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDRECT

  DDERR_INVALIDSURFACETYPE

  DDERR_NOSTRETCHHW

  DDERR_NOTAOVERLAYSURFACE

  DDERR_SURFACELOST

  DDERR_UNSUPPORTED

  DDERR_XALIGN

lpSrcRect
Address of a **RECT** structure that defines the x, y, width, and height of the region on the source surface being used as the overlay.

lpDDDestSurface
Address of the DirectDraw surface that is being overlaid.

lpDestRect
Address of a **RECT** structure that defines the x, y, width, and height of the region on the destination surface that the overlay should be moved to.

dwFlags
DDOVER_ADDDIRTYRECT
Adds a dirty rectangle to an emulated overlaid surface.

DDOVER_ALPHADEST

Uses either the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for this overlay.

DDOVER_ALPHADESTCONSTOVERRIDE

Uses the **dwAlphaDestConst** member of the **DDOVERLAYFX** structure as the destination alpha channel for this overlay.

DDOVER_ALPHADESTNEG

Indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_ALPHADESTSURFACEOVERRIDE

Uses the **lpDDSAlphaDest** member of the **DDOVERLAYFX** structure as the alpha channel destination for this overlay.

DDOVER_ALPHAEDGEBLEND

Uses the **dwAlphaEdgeBlend** member of the **DDOVERLAYFX** structure as the alpha channel for the edges of the image that border the color key colors.

DDOVER_ALPHASRC

Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the source alpha channel for this overlay.

DDOVER_ALPHASRCCONSTOVERRIDE

Uses the **dwAlphaSrcConst** member of the **DDOVERLAYFX** structure as the source alpha channel for this overlay.

DDOVER_ALPHASRCNEG

Indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_ALPHASRCSURFACEOVERRIDE

Uses the **lpDDSAlphaSrc** member of the **DDOVERLAYFX** structure as the alpha channel source for this overlay.

DDOVER_DDFX

Uses the overlay FX flags to define special overlay effects.

DDOVER_HIDE

Turns this overlay off.

DDOVER_KEYDEST

Uses the color key associated with the destination surface.

DDOVER_KEYDESTOVERRIDE

Uses the **dckDestColorkey** member of the **DDOVERLAYFX** structure as the color key for the destination surface.

DDOVER_KEYSRC

Uses the color key associated with the source surface.

DDOVER_KEYSRCOVERRIDE

Uses the **dckSrcColorkey** member of the **DDOVERLAYFX** structure as the color key for the source surface.

DDOVER_SHOW

Turns this overlay on.

DDOVER_ZORDER

Uses the **dwZOrderFlags** member of the **DDOVERLAYFX** structure as the z-order for the display of this overlay. The **lpDDSRelative** member is used if the **dwZOrderFlags** member is set to either DDOVERZ_INSERTINBACKOF or DDOVERZ_INSERTINFRONTOF.

lpDDOverlayFx

See the **DDOVERLAYFX** structure.

IDirectDrawSurface2::UpdateOverlayDisplay

```
HRESULT UpdateOverlayDisplay(DWORD dwFlags);
```

Repaints the rectangles in the dirty rectangle list of all active overlays. This clears the dirty rectangle list. This method is for software emulation only—it does nothing if the hardware supports overlays.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_INVALIDSURFACETYPE

  DDERR_UNSUPPORTED

dwFlags

Type of update to perform. One of the following flags:

DDOVER_REFRESHDIRTYRECTS

Updates the overlay display using the list of dirty rectangles previously constructed for this destination. This clears the dirty rectangle list.

DDOVER_REFRESHALL

Ignores the dirty rectangle list and updates the overlay display completely. This clears the dirty rectangle list.

See also **IDirectDrawSurface2::AddOverlayDirtyRect**

IDirectDrawSurface2::UpdateOverlayZOrder

```
HRESULT UpdateOverlayZOrder(DWORD dwFlags,
    LPDIRECTDRAWSURFACE2 lpDDSReference);
```

Sets the z-order of an overlay.

- Returns DD_OK if successful, or one of the following error values otherwise:

  DDERR_INVALIDOBJECT

  DDERR_INVALIDPARAMS

  DDERR_NOTAOVERLAYSURFACE

dwFlags

One of the following flags:

DDOVERZ_INSERTINBACKOF

Inserts this overlay in the overlay chain behind the reference overlay.

DDOVERZ_INSERTINFRONTOF

Inserts this overlay in the overlay chain in front of the reference overlay.

DDOVERZ_MOVEBACKWARD

Moves this overlay one position backward in the overlay chain.

DDOVERZ_MOVEFORWARD

Moves this overlay one position forward in the overlay chain.

DDOVERZ_SENDTOBACK

Moves this overlay to the back of the overlay chain.

DDOVERZ_SENDTOFRONT

Moves this overlay to the front of the overlay chain.

lpDDSReference

Address of the DirectDraw surface to be used as a relative position in the overlay chain. This parameter is needed only for DDOVERZ_INSERTINBACKOF and DDOVERZ_INSERTINFRONTOF.

See also **IDirectDrawSurface2::EnumOverlayZOrders**

Structures

DDBLTBATCH

```
typedef struct _DDBLTBATCH{
    LPRECT              lprDest;
    LPDIRECTDRAWSURFACE lpDDSSrc;
    LPRECT              lprSrc;
    DWORD               dwFlags;
```

```
    LPDDBLTFX           lpDDBltFx;
} DDBLTBATCH,FAR *LPDDBLTBATCH;
```

Passes blit operations to the **IDirectDrawSurface2::BltBatch** method.

lprDest
Address of a **RECT** structure that defines the destination for the blit.

lpDDSSrc
Address of a DirectDrawSurface object that will be the source of the blit.

lprSrc
Address of a **RECT** structure that defines the source rectangle of the blit.

dwFlags
Optional control flags.

DDBLT_ALPHADEST

Uses either the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for this blit.

DDBLT_ALPHADESTCONSTOVERRIDE

Uses the **dwAlphaDestConst** member of the **DDBLTFX** structure as the alpha channel for the destination surface for this blit.

DDBLT_ALPHADESTNEG

Indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHADESTSURFACEOVERRIDE

Uses the **lpDDSAlphaDest** member of the **DDBLTFX** structure as the alpha channel for the destination surface for this blit.

DDBLT_ALPHAEDGEBLEND

Uses the **dwAlphaEdgeBlend** member of the **DDBLTFX** structure as the alpha channel for the edges of the image that border the color key colors.

DDBLT_ALPHASRC

Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the alpha channel for this blit.

DDBLT_ALPHASRCCONSTOVERRIDE

Uses the **dwAlphaSrcConst** member of the **DDBLTFX** structure as the source alpha channel for this blit.

DDBLT_ALPHASRCNEG

Indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHASRCSURFACEOVERRIDE

Uses the **lpDDSAlphaSrc** member of the **DDBLTFX** structure as the alpha channel source for this blit.

DDBLT_ASYNC

Processes this blit asynchronously through the FIFO hardware in the order received. If there is no room in the FIFO hardware, the call fails.

DDBLT_COLORFILL

Uses the **dwFillColor** member of the **DDBLTFX** structure as the RGB color that fills the destination rectangle on the destination surface.

DDBLT_DDFX

Uses the **dwDDFX** member of the **DDBLTFX** structure to specify the effects to be used for this blit.

DDBLT_DDROPS

Uses the **dwDDROPS** member of the **DDBLTFX** structure to specify the raster operations (ROPs) that are not part of the Win32 API.

DDBLT_KEYDEST

Uses the color key associated with the destination surface.

DDBLT_KEYDESTOVERRIDE

Uses the **dckDestColorkey** member of the **DDBLTFX** structure as the color key for the destination surface.

DDBLT_KEYSRC

Uses the color key associated with the source surface.

DDBLT_KEYSRCOVERRIDE

Uses the **dckSrcColorkey** member of the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP

Uses the **dwROP** member of the **DDBLTFX** structure for the ROP for this blit. The ROPs are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE

Uses the **dwRotationAngle** member of the **DDBLTFX** structure as the rotation angle (specified in 1/100th of a degree) for the surface.

DDBLT_ZBUFFER

Performs a z-buffered blit using the z-buffers attached to the source and destination surfaces and the **dwZBufferOpCode** member of the **DDBLTFX** structure as the z-buffer opcode.

DDBLT_ZBUFFERDESTCONSTOVERRIDE

Performs a z-buffered blit using the **dwZDestConst** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the destination.

DDBLT_ZBUFFERDESTOVERRIDE

Performs a z-buffered blit using the **lpDDSZBufferDest** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the destination.

DDBLT_ZBUFFERSRCCONSTOVERRIDE

Performs a z-buffered blit using the **dwZSrcConst** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the source.

DDBLT_ZBUFFERSRCOVERRIDE

A z-buffered blit using the **lpDDSZBufferSrc** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the source.

lpDDBltFx

Address of a **DDBLTFX** structure specifying additional blit effects.

DDBLTFX

```
typedef struct _DDBLTFX{
    DWORD dwSize;
    DWORD dwDDFX;
    DWORD dwROP;
    DWORD dwDDROP;
    DWORD dwRotationAngle;
    DWORD dwZBufferOpCode;
    DWORD dwZBufferLow;
    DWORD dwZBufferHigh;
    DWORD dwZBufferBaseDest;
    DWORD dwZDestConstBitDepth;
union
{
        DWORD               dwZDestConst;
        LPDIRECTDRAWSURFACE lpDDSZBufferDest;
};
    DWORD dwZSrcConstBitDepth;
union
{
        DWORD               dwZSrcConst;
        LPDIRECTDRAWSURFACE lpDDSZBufferSrc;

};
    DWORD dwAlphaEdgeBlendBitDepth;
    DWORD dwAlphaEdgeBlend;
    DWORD dwReserved;
    DWORD dwAlphaDestConstBitDepth;
union
{
        DWORD               dwAlphaDestConst;
        LPDIRECTDRAWSURFACE lpDDSAlphaDest;
};
    DWORD dwAlphaSrcConstBitDepth;
union
{
```

```
        DWORD               dwAlphaSrcConst;
        LPDIRECTDRAWSURFACE lpDDSAlphaSrc;
};
union
{
        DWORD               dwFillColor;
        DWORD               dwFillDepth;
        LPDIRECTDRAWSURFACE lpDDSPattern;
};
DDCOLORKEY ddckDestColorkey;
DDCOLORKEY ddckSrcColorkey;
} DDBLTFX,FAR* LPDDBLTFX;
```

Passes raster operations, effects, and override information to the **IDirectDrawSurface2::Blt** method. This structure is also part of the **DDBLTBATCH** structure used with the **IDirectDrawSurface2::BltBatch** method.

dwSize

Size of the structure. This member must be initialized before the structure is used.

dwDDFX

Type of FX operations.

DDBLTFX_ARITHSTRETCHY

Uses arithmetic stretching along the y-axis for this blit.

DDBLTFX_MIRRORLEFTRIGHT

Turns the surface on its y-axis. This blit mirrors the surface from left to right.

DDBLTFX_MIRRORUPDOWN

Turns the surface on its x-axis. This blit mirrors the surface from top to bottom.

DDBLTFX_NOTEARING

Schedules this blit to avoid tearing.

DDBLTFX_ROTATE180

Rotates the surface 180 degrees clockwise during this blit.

DDBLTFX_ROTATE270

Rotates the surface 270 degrees clockwise during this blit.

DDBLTFX_ROTATE90

Rotates the surface 90 degrees clockwise during this blit.

DDBLTFX_ZBUFFERBASEDEST

Adds the **dwZBufferBaseDest** member to each of the source z-values before comparing them with the destination z-values during this z-blit.

DDBLTFX_ZBUFFERRANGE

Uses the **dwZBufferLow** and **dwZBufferHigh** members as range values to specify limits to the bits copied from a source surface during this z-blit.

dwROP
Win32 raster operations.

dwDDROP
DirectDraw raster operations.

dwRotationAngle
Rotation angle for the blit.

dwZBufferOpCode
Z-buffer compares.

dwZBufferLow
Low limit of a z-buffer.

dwZBufferHigh
High limit of a z-buffer.

dwZBufferBaseDest
Destination base value of a z-buffer.

dwZDestConstBitDepth
Bit depth of the destination z-constant.

dwZDestConst
Constant used as the z-buffer destination.

lpDDSZBufferDest
Surface used as the z-buffer destination.

dwZSrcConstBitDepth
Bit depth of the source z-constant.

dwZSrcConst
Constant used as the z-buffer source.

lpDDSZBufferSrc
Surface used as the z-buffer source.

dwAlphaEdgeBlendBitDepth
Bit depth of the constant for an alpha edge blend.

dwAlphaEdgeBlend
Alpha constant used for edge blending.

dwReserved
Reserved for future use.

dwAlphaDestConstBitDepth
Bit depth of the destination alpha constant.

dwAlphaDestConst
Constant used as the alpha channel destination.

lpDDSAlphaDest
Surface used as the alpha channel destination.

dwAlphaSrcConstBitDepth
Bit depth of the source alpha constant.

dwAlphaSrcConst
Constant used as the alpha channel source.

lpDDSAlphaSrc
Surface used as the alpha channel source.

dwFillColor
Color used to fill a surface when DDBLT_COLORFILL is specified. This value can be either an RGB triple or a palette index, depending on the surface type.

dwFillDepth
Depth value for the z-buffer.

lpDDSPattern
Surface to use as a pattern. The pattern can be used in certain blit operations that combine a source and a destination.

ddckDestColorkey
Destination color key override.

ddckSrcColorkey
Source color key override.

DDCAPS

```
typedef struct _DDCAPS{
    DWORD    dwSize;
    DWORD    dwCaps;
    DWORD    dwCaps2;
    DWORD    dwCKeyCaps;
    DWORD    dwFXCaps;
    DWORD    dwFXAlphaCaps;
    DWORD    dwPalCaps;
    DWORD    dwSVCaps;
    DWORD    dwAlphaBltConstBitDepths;
    DWORD    dwAlphaBltPixelBitDepths;
    DWORD    dwAlphaBltSurfaceBitDepths;
    DWORD    dwAlphaOverlayConstBitDepths;
    DWORD    dwAlphaOverlayPixelBitDepths;
    DWORD    dwAlphaOverlaySurfaceBitDepths;
    DWORD    dwZBufferBitDepths;

    DWORD    dwVidMemTotal;
    DWORD    dwVidMemFree;
    DWORD    dwMaxVisibleOverlays;
    DWORD    dwCurrVisibleOverlays;
    DWORD    dwNumFourCCCodes;
    DWORD    dwAlignBoundarySrc;
    DWORD    dwAlignSizeSrc;
    DWORD    dwAlignBoundaryDest;
```

```
    DWORD    dwAlignSizeDest;
    DWORD    dwAlignStrideAlign;
    DWORD    dwRops[DD_ROP_SPACE];
    DDSCAPS  ddsCaps;
    DWORD    dwMinOverlayStretch;
    DWORD    dwMaxOverlayStretch;
    DWORD    dwMinLiveVideoStretch;

    DWORD    dwMaxLiveVideoStretch;
    DWORD    dwMinHwCodecStretch;
    DWORD    dwMaxHwCodecStretch;
    DWORD    dwReserved1;
    DWORD    dwReserved2;
    DWORD    dwReserved3;
    DWORD    dwSVBCaps;
    DWORD    dwSVBCKeyCaps;
    DWORD    dwSVBFXCaps;
    DWORD    dwSVBRops[DD_ROP_SPACE];
    DWORD    dwVSBCaps;
    DWORD    dwVSBCKeyCaps;
    DWORD    dwVSBFXCaps;
    DWORD    dwVSBRops[DD_ROP_SPACE];
    DWORD    dwSSBCaps;
    DWORD    dwSSBCKeyCaps;

    DWORD    dwSSBCFXCaps;
    DWORD    dwSSBRops[DD_ROP_SPACE];
    DWORD    dwReserved4;
    DWORD    dwReserved5;
    DWORD    dwReserved6;

} DDCAPS,FAR* LPDDCAPS;
```

Represents the capabilities of the hardware exposed through the DirectDraw object. This structure contains a **DDSCAPS** structure used in this context to describe what kinds of DirectDrawSurface objects can be created. It may not be possible to simultaneously create all of the surfaces described by these capabilities. This structure is used with the **IDirectDraw2::GetCaps** and **IDirectDrawPalette::GetCaps** methods.

dwSize
Size of the structure. This member must be initialized before the structure is used.

dwCaps
Driver-specific capabilities.

DDCAPS_3D
Indicates that the display hardware has 3D acceleration.

DDCAPS_ALIGNBOUNDARYDEST

Indicates that DirectDraw will support only those source rectangles with the x-axis aligned to the **dwAlignBoundaryDest** boundaries of the surface.

DDCAPS_ALIGNBOUNDARYSRC

Indicates that DirectDraw will support only those source rectangles with the x-axis aligned to the **dwAlignBoundarySrc** boundaries of the surface.

DDCAPS_ALIGNSIZEDEST

Indicates that DirectDraw will support only those source rectangles whose x-axis sizes, in bytes, are **dwAlignSizeDest** multiples.

DDCAPS_ALIGNSIZESRC

Indicates that DirectDraw will support only those source rectangles whose x-axis sizes, in bytes, are **dwAlignSizeSrc** multiples.

DDCAPS_ALIGNSTRIDE

Indicates that DirectDraw will create display memory surfaces that have a stride alignment equal to the **dwAlignStrideAlign** value.

DDCAPS_ALPHA

Indicates that the display hardware supports an alpha channel during blit operations.

DDCAPS_BANKSWITCHED

Indicates that the display hardware is bank-switched and is potentially very slow at random access to display memory.

DDCAPS_BLT

Indicates that display hardware is capable of blit operations.

DDCAPS_BLTCOLORFILL

Indicates that display hardware is capable of color filling with a blitter.

DDCAPS_BLTDEPTHFILL

Indicates that display hardware is capable of depth filling z-buffers with a blitter.

DDCAPS_BLTFOURCC

Indicates that display hardware is capable of color-space conversions during blit operations.

DDCAPS_BLTQUEUE

Indicates that display hardware is capable of asynchronous blit operations.

DDCAPS_BLTSTRETCH

Indicates that display hardware is capable of stretching during blit operations.

DDCAPS_CANBLTSYSMEM

Indicates that display hardware is capable of blitting to or from system memory.

DDCAPS_CANCLIP

Indicates that display hardware is capable of clipping with blitting.

DDCAPS_CANCLIPSTRETCHED

Indicates that display hardware is capable of clipping while stretch blitting.

DDCAPS_COLORKEY

Supports some form of color key in either overlay or blit operations. More specific color key capability information can be found in the **dwCKeyCaps** member.

DDCAPS_COLORKEYHWASSIST

Indicates that the color key is hardware assisted.

DDCAPS_GDI

Indicates that display hardware is shared with GDI.

DDCAPS_NOHARDWARE

Indicates that there is no hardware support.

DDCAPS_OVERLAY

Indicates that display hardware supports overlays.

DDCAPS_OVERLAYCANTCLIP

Indicates that display hardware supports overlays but cannot clip them.

DDCAPS_OVERLAYFOURCC

Indicates that overlay hardware is capable of color-space conversions during overlay operations.

DDCAPS_OVERLAYSTRETCH

Indicates that overlay hardware is capable of stretching.

DDCAPS_PALETTE

Indicates that DirectDraw is capable of creating and supporting DirectDrawPalette objects for more surfaces than only the primary surface.

DDCAPS_PALETTEVSYNC

Indicates that DirectDraw is capable of updating a palette synchronized with the vertical refresh.

DDCAPS_READSCANLINE

Indicates that display hardware is capable of returning the current scanline.

DDCAPS_STEREOVIEW

Indicates that display hardware has stereo vision capabilities.

DDCAPS_VBI

Indicates that display hardware is capable of generating a vertical-blank interrupt.

DDCAPS_ZBLTS

Supports the use of z-buffers with blit operations.

DDCAPS_ZOVERLAYS

Supports the use of the **IDirectDrawSurface2::UpdateOverlayZOrder** method as a z-value for overlays to control their layering.

dwCaps2

More driver-specific capabilities.

DDCAPS2_CERTIFIED

Indicates that display hardware is certified.

DDCAPS2_NO2DDURING3DSCENE

Indicates that 2D operations such as **IDirectDrawSurface2::Blt** and **IDirectDrawSurface2::Lock** cannot be performed on any surfaces that Direct3D is using between calls to the **IDirect3DDevice::BeginScene** and **IDirect3DDevice::EndScene** methods.

dwCKeyCaps

Color-key capabilities.

DDCKEYCAPS_DESTBLT

Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACE

Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACEYUV

Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTBLTYUV

Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTOVERLAY

Supports overlaying with color keying of the replaceable bits of the destination surface being overlaid for RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACE

Supports a color space as the color key for the destination of RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV

Supports a color space as the color key for the destination of YUV colors.

DDCKEYCAPS_DESTOVERLAYONEACTIVE

Supports only one active destination color key value for visible overlay surfaces.

DDCKEYCAPS_DESTOVERLAYYUV

Supports overlaying using color keying of the replaceable bits of the destination surface being overlaid for YUV colors.

DDCKEYCAPS_NOCOSTOVERLAY

Indicates there are no bandwidth trade-offs for using the color key with an overlay.

DDCKEYCAPS_SRCBLT

Supports transparent blitting using the color key for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACE

Supports transparent blitting using a color space for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACEYUV

Supports transparent blitting using a color space for the source with this surface for YUV colors.

DDCKEYCAPS_SRCBLTYUV

Supports transparent blitting using the color key for the source with this surface for YUV colors.

DDCKEYCAPS_SRCOVERLAY

Supports overlaying using the color key for the source with this overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACE

Supports overlaying using a color space as the source color key for the overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV

Supports overlaying using a color space as the source color key for the overlay surface for YUV colors.

DDCKEYCAPS_SRCOVERLAYONEACTIVE

Supports only one active source color key value for visible overlay surfaces.

DDCKEYCAPS_SRCOVERLAYYUV

Supports overlaying using the color key for the source with this overlay surface for YUV colors.

dwFXCaps

Driver-specific stretching and effects capabilities.

DDFXCAPS_BLTARITHSTRETCHY

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-

axis (vertically).

DDFXCAPS_BLTARITHSTRETCHYN

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_BLTMIRRORLEFTRIGHT

Supports mirroring left to right in a blit operation.

DDFXCAPS_BLTMIRRORUPDOWN

Supports mirroring top to bottom in a blit operation.

DDFXCAPS_BLTROTATION

Supports arbitrary rotation in a blit operation.

DDFXCAPS_BLTROTATION90

Supports 90-degree rotations in a blit operation.

DDFXCAPS_BLTSHRINKX

Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKXN

Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKY

Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKYN

Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHX

Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHXN

Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHY

Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHYN

Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_OVERLAYARITHSTRETCHY

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during an overlay operation. Occurs along the y-axis (vertically).

DDFXCAPS_OVERLAYARITHSTRETCHYN

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during an overlay operation. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_OVERLAYMIRRORLEFTRIGHT

Supports mirroring of overlays around the vertical axis.

DDFXCAPS_OVERLAYMIRRORUPDOWN

Supports mirroring of overlays across the horizontal axis.

DDFXCAPS_OVERLAYSHRINKX

Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKXN

Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKY

Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKYN

Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSTRETCHX

Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHXN

Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHY

Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHYN

Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

dwFXAlphaCaps

Driver-specific alpha capabilities.

DDFXALPHACAPS_BLTALPHAEDGEBLEND

Supports alpha blending around the edge of a source color-keyed surface. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELS

Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELSNEG

Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA is set. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACES

Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACESNEG

Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be set only if DDFXCAPS_ALPHASURFACES has been set. Used for blit operations.

DDFXALPHACAPS_OVERLAYALPHAEDGEBLEND

Supports alpha blending around the edge of a source color-keyed surface. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELS

Supports alpha information in pixel format. The bit depth of alpha

information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELSNEG

Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDFXCAPS_ALPHAPIXELS has been set. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACES

Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACESNEG

Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDFXCAPS_ALPHASURFACES has been set. Used for overlays.

dwPalCaps

Palette capabilities.

DDPCAPS_1BIT

Indicates that the index is 1 bit. There are two entries in the color table.

DDPCAPS_2BIT

Indicates that the index is 2 bits. There are four entries in the color table.

DDPCAPS_4BIT

Indicates that the index is 4 bits. There are 16 entries in the color table.

DDPCAPS_8BIT

Indicates that the index is 8 bits. There are 256 entries in the color table.

DDPCAPS_8BITENTRIES

Specifies an index to an 8-bit color index. This field is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT capability and when the target surface is in 8 bits per pixel (bpp). Each color entry is 1 byte long and is an index to an 8-bpp palette on the destination surface.

DDPCAPS_ALLOW256

Indicates that this palette can have all 256 entries defined.

DDPCAPS_PRIMARYSURFACE

Indicates that the palette is attached to the primary surface. Changing the palette has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_PRIMARYSURFACELEFT

Indicates that the palette is attached to the primary surface on the left. Changing the palette has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_VSYNC

Indicates that the palette can be modified synchronously with the monitor's refresh rate.

dwSVCaps

Stereo vision capabilities.

DDSVCAPS_ENIGMA

Indicates that the stereo view is accomplished using Enigma encoding.

DDSVCAPS_FLICKER

Indicates that the stereo view is accomplished using high-frequency flickering.

DDSVCAPS_REDBLUE

Indicates that the stereo view is accomplished when the viewer looks at the image through red and blue filters placed over the left and right eyes. All images must adapt their color spaces for this process.

DDSVCAPS_SPLIT

Indicates that the stereo view is accomplished with split-screen technology.

dwAlphaBltConstBitDepths

DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

dwAlphaBltPixelBitDepths

DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaBltSurfaceBitDepths

DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlayConstBitDepths

DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlayPixelBitDepths

DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlaySurfaceBitDepths

DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwZBufferBitDepths
DDBD_8, DDBD_16, DDBD_24, or DDBD_32. (Indicates 8-, 16-, 24-, or 32-bits per pixel.)

dwVidMemTotal
Total amount of display memory.

dwVidMemFree
Amount of free display memory.

dwMaxVisibleOverlays
Maximum number of visible overlays.

dwCurrVisibleOverlays
Current number of visible overlays.

dwNumFourCCCodes
Number of FourCC codes.

dwAlignBoundarySrc
Source rectangle alignment.

dwAlignSizeSrc
Source rectangle byte size.

dwAlignBoundaryDest
Destination rectangle alignment.

dwAlignSizeDest
Destination rectangle byte size.

dwAlignStrideAlign
Stride alignment.

dwRops[DD_ROP_SPACE]
Raster operations supported.

ddsCaps
DDSCAPS structure with general capabilities.

dwMinOverlayStretch and **dwMaxOverlayStretch**
Minimum and maximum overlay stretch factors multiplied by 1000. For example, 1.3 = 1300.

dwMinLiveVideoStretch and **dwMaxLiveVideoStretch**
Minimum and maximum live video stretch factors multiplied by 1000. For example, 1.3 = 1300.

dwMinHwCodecStretch and **dwMaxHwCodecStretch**
Minimum and maximum hardware codec stretch factors multiplied by 1000. For example, 1.3 = 1300.

dwReserved1, **dwReserved2**, and **dwReserved3**
Reserved for future use.

dwSVBCaps
Driver-specific capabilities for system-memory-to-display-memory blits.

dwSVBCKeyCaps
Driver color-key capabilities for system-memory-to-display-memory blits.

dwSVBFXCaps
Driver FX capabilities for system-memory-to-display-memory blits.

dwSVBRops[DD_ROP_SPACE]
Raster operations supported for system-memory-to-display-memory blits.

dwVSBCaps
Driver-specific capabilities for display-memory-to-system-memory blits.

dwVSBCKeyCaps
Driver color-key capabilities for display-memory-to-system-memory blits.

dwVSBFXCaps
Driver FX capabilities for display-memory-to-system-memory blits.

dwVSBRops[DD_ROP_SPACE]
Supports raster operations for display-memory-to-system-memory blits.

dwSSBCaps
Driver-specific capabilities for system-memory-to-system-memory blits.

dwSSBCKeyCaps
Driver color-key capabilities for system-memory-to-system-memory blits.

dwSSBCFXCaps
Driver FX capabilities for system-memory-to-system-memory blits.

dwSSBRops[DD_ROP_SPACE]
Raster operations supported for system-memory-to-system-memory blits.

dwReserved4, **dwReserved5**, and **dwReserved6**
Reserved for future use.

DDCOLORKEY

```
typedef struct _DDCOLORKEY{
    DWORD dwColorSpaceLowValue;
    DWORD dwColorSpaceHighValue;
} DDCOLORKEY,FAR* LPDDCOLORKEY;
```

Describes a source color key, destination color key, or color space. A color key is specified if the low and high range values are the same. This structure is used with the **IDirectDrawSurface2::GetColorKey** and **IDirectDrawSurface2::SetColorKey** methods.

dwColorSpaceLowValue
Low value, inclusive, of the color range that is to be used as the color key.

dwColorSpaceHighValue
High value, inclusive, of the color range that is to be used as the color key.

DDOVERLAYFX

```
typedef struct _DDOVERLAYFX{
    DWORD dwSize;
    DWORD dwAlphaEdgeBlendBitDepth;
    DWORD dwAlphaEdgeBlend;
    DWORD dwReserved;
    DWORD dwAlphaDestConstBitDepth;
union
{
        DWORD               dwAlphaDestConst;
        LPDIRECTDRAWSURFACE lpDDSAlphaDest;
};
    DWORD dwAlphaSrcConstBitDepth;
union
{
        DWORD               dwAlphaSrcConst;
        LPDIRECTDRAWSURFACE lpDDSAlphaSrc;
};
    DDCOLORKEY dckDestColorkey;
    DDCOLORKEY dckSrcColorkey;

    DWORD dwDDFX;
    DWORD dwFlags;
} DDOVERLAYFX,FAR *LPDDOVERLAYFX;
```

Passes override information to the **IDirectDrawSurface2::UpdateOverlay** method.

dwSize
Size of the structure. This members must be initialized before the structure is used.

dwAlphaEdgeBlendBitDepth
Bit depth used to specify the constant for an alpha edge blend.

dwAlphaEdgeBlend
Constant to use as the alpha for an edge blend.

dwReserved
Reserved for future use.

dwAlphaDestConstBitDepth
Bit depth used to specify the alpha constant for a destination.

dwAlphaDestConst
Constant to use as the alpha channel for a destination.

lpDDSAlphaDest
Address of a surface to use as the alpha channel for a destination.

dwAlphaSrcConstBitDepth
Bit depth used to specify the alpha constant for a source.

dwAlphaSrcConst
Constant to use as the alpha channel for a source.

lpDDSAlphaSrc
Address of a surface to use as the alpha channel for a source.

dckDestColorkey
Destination color key override.

dckSrcColorkey
Source color key override.

dwDDFX
Overlay FX flags.

DDOVERFX_ARITHSTRETCHY
If stretching, use arithmetic stretching along the y-axis for this overlay.

DDOVERFX_MIRRORLEFTRIGHT
Mirror the overlay around the vertical axis.

DDOVERFX_MIRRORUPDOWN
Mirror the overlay around the horizontal axis.

dwFlags
This member is currently not used and must be set to 0.

DDPIXELFORMAT

```
typedef struct _DDPIXELFORMAT{
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwFourCC;
union
{
        DWORD dwRGBBitCount;
        DWORD dwYUVBitCount;
        DWORD dwZBufferBitDepth;
        DWORD dwAlphaBitDepth;
};
union
{
        DWORD dwRBitMask;
        DWORD dwYBitMask;
};
union
{
        DWORD dwGBitMask;
        DWORD dwUBitMask;
};
union
{
```

```
        DWORD dwBBitMask;
        DWORD dwVBitMask;
};
union
{
        DWORD dwRGBAlphaBitMask;

        DWORD dwYUVAlphaBitMask;
};
} DDPIXELFORMAT, FAR* LPDDPIXELFORMAT;
```

Describes the pixel format of a DirectDrawSurface object for the **IDirectDrawSurface2::GetPixelFormat** method.

dwSize
Size of the structure. This member must be initialized before the structure is used.

dwFlags
Optional control flags.

DDPF_ALPHA
The pixel format describes an alpha-only surface.

DDPF_ALPHAPIXELS
The surface has alpha channel information in the pixel format.

DDPF_COMPRESSED
The surface will accept pixel data in the specified format and compress it during the write operation.

DDPF_FOURCC
The FourCC code is valid.

DDPF_PALETTEINDEXED1

DDPF_PALETTEINDEXED2

DDPF_PALETTEINDEXED4

DDPF_PALETTEINDEXED8
The surface is 1-, 2-, 4-, or 8-bit color indexed.

DDPF_PALETTEINDEXEDTO8
The surface is 1-, 2-, or 4-bit color indexed to an 8-bit palette.

DDPF_RGB
The RGB data in the pixel format structure is valid.

DDPF_RGBTOYUV
The surface will accept RGB data and translate it during the write operation to YUV data. The format of the data to be written will be contained in the pixel format structure. The DDPF_RGB flag will be set.

DDPF_YUV

The YUV data in the pixel format structure is valid.

DDPF_ZBUFFER

The pixel format describes a z-buffer-only surface.

dwFourCC
FourCC code.

dwRGBBitCount
RGB bits per pixel (4, 8, 16, 24, or 32).

dwYUVBitCount
YUV bits per pixel (DDBD_4, DDBD_8, DDBD_16, DDBD_24, or DDBD_32).

dwZBufferBitDepth
Z-buffer bit depth (8, 16, 24, or 32).

dwAlphaBitDepth
Alpha channel bit depth (DDBD_1, DDBD_2, DDBD_4, or DDBD_8).

dwRBitMask
Mask for red bits.

dwYBitMask
Mask for Y bits.

dwGBitMask
Mask for green bits.

dwUBitMask
Mask for U bits.

dwBBitMask
Mask for blue bits.

dwVBitMask
Mask for V bits.

dwRGBAlphaBitMask
Mask for alpha channel.

dwYUVAlphaBitMask
Mask for alpha channel.

DDSCAPS

```
typedef struct _DDSCAPS{
        DWORD dwCaps;
} DDSCAPS,FAR* LPDDSCAPS;
```

Defines the capabilities of a DirectDrawSurface object. This structure is part of the **DDCAPS** structure that is used to describe the capabilities of the DirectDraw object.

dwCaps
Capabilities of the surface. One or more of the following flags:

DDSCAPS_3D

Supported for backward compatibility. Applications should use the DDSCAPS_3DDEVICE flag, instead.

DDSCAPS_3DDEVICE

Indicates that this surface can be used for 3D rendering. Applications can use this flag to ensure that a device that can only render to a certain heap has off-screen surfaces allocated from the correct heap. If this flag is set for a heap, the surface is not allocated from that heap.

DDSCAPS_ALLOCONLOAD

Indicates that memory for the surface is not allocated until the surface is loaded by using the **IDirect3DTexture::Load** method.

DDSCAPS_ALPHA

Indicates that this surface contains alpha information. The pixel format must be queried to determine whether this surface contains only alpha information or alpha information interlaced with pixel color data (such as RGBA or YUVA).

DDSCAPS_BACKBUFFER

Indicates that this surface is the back buffer of a surface flipping structure. Typically, this capability is set by the **IDirectDraw2::CreateSurface** method when the DDSCAPS_FLIP flag is used. Only the surface that immediately precedes the DDSCAPS_FRONTBUFFER surface has this capability set. The other surfaces are identified as back buffers by the presence of the DDSCAPS_FLIP flag, their attachment order, and the absence of the DDSCAPS_FRONTBUFFER and DDSCAPS_BACKBUFFER capabilities. If this capability is sent to the **IDirectDraw2::CreateSurface** method, a standalone back buffer is being created. After this method is called, this surface could be attached to a front buffer, another back buffer, or both to form a flipping surface structure. For more information, see **IDirectDrawSurface2::AddAttachedSurface**. DirectDraw supports an arbitrary number of surfaces in a flipping structure.

DDSCAPS_COMPLEX

Indicates that a complex surface is being described. A complex surface results in the creation of more than one surface. The additional surfaces are attached to the root surface. The complex structure can be destroyed only by destroying the root.

DDSCAPS_FLIP

Indicates that this surface is a part of a surface flipping structure. When this capability is passed to the **IDirectDraw2::CreateSurface** method, a front buffer and one or more back buffers are created. DirectDraw sets the DDSCAPS_FRONTBUFFER bit on the front-buffer surface and the DDSCAPS_BACKBUFFER bit on the surface adjacent to the front-buffer surface. The **dwBackBufferCount** member of the **DDSURFACEDESC** structure must be set to at least 1 in order for the method call to succeed. The DDSCAPS_COMPLEX capability must always be set when creating

multiple surfaces by using the **IDirectDraw2::CreateSurface** method.

DDSCAPS_FRONTBUFFER

Indicates that this surface is the front buffer of a surface flipping structure. This flag is typically set by the **IDirectDraw2::CreateSurface** method when the DDSCAPS_FLIP capability is set. If this capability is sent to the **IDirectDraw2::CreateSurface** method, a standalone front buffer is created. This surface will not have the DDSCAPS_FLIP capability. It can be attached to other back buffers to form a flipping structure by using **IDirectDrawSurface2::AddAttachedSurface**.

DDSCAPS_HWCODEC

Indicates that this surface should be able to have a stream decompressed to it by the hardware.

DDSCAPS_LIVEVIDEO

Indicates that this surface should be able to receive live video.

DDSCAPS_MIPMAP

Indicates that this surface is one level of a mipmap. This surface will be attached to other DDSCAPS_MIPMAP surfaces to form the mipmap. This can be done explicitly by creating a number of surfaces and attaching them by using the **IDirectDrawSurface2::AddAttachedSurface** method, or implicitly by the **IDirectDraw2::CreateSurface** method. If this capability is set, DDSCAPS_TEXTURE must also be set.

DDSCAPS_MODEX

Indicates that this surface is a 320×200 or 320×240 Mode X surface.

DDSCAPS_OFFSCREENPLAIN

Indicates that this surface is any off-screen surface that is not an overlay, texture, z-buffer, front-buffer, back-buffer, or alpha surface. It is used to identify plain surfaces.

DDSCAPS_OVERLAY

Indicates that this surface is an overlay. It may or may not be directly visible depending on whether it is currently being overlaid onto the primary surface. DDSCAPS_VISIBLE can be used to determine if it is being overlaid at the moment.

DDSCAPS_OWNDC

Indicates that this surface will have a device context (DC) association for a long period.

DDSCAPS_PALETTE

Indicates that this device driver allows unique DirectDrawPalette objects to be created and attached to this surface.

DDSCAPS_PRIMARYSURFACE

Indicates the surface is the primary surface. It represents what is visible to the user at the moment.

DDSCAPS_PRIMARYSURFACELEFT

Indicates that this surface is the primary surface for the left eye. It represents what is visible to the user's left eye at the moment. When this surface is created, the surface with the DDSCAPS_PRIMARYSURFACE capability represents what is seen by the user's right eye.

DDSCAPS_SYSTEMMEMORY

Indicates that this surface memory was allocated in system memory.

DDSCAPS_TEXTURE

Indicates that this surface can be used as a 3D texture. It does not indicate whether the surface is being used for that purpose.

DDSCAPS_VIDEOMEMORY

Indicates that this surface exists in display memory.

DDSCAPS_VISIBLE

Indicates that changes made to this surface are immediately visible. It is always set for the primary surface, as well as for overlays while they are being overlaid and texture maps while they are being textured.

DDSCAPS_WRITEONLY

Indicates that only write access is permitted to the surface. Read access from the surface may generate a general protection (GP) fault, but the read results from this surface will not be meaningful.

DDSCAPS_ZBUFFER

Indicates that this surface is the z-buffer. The z-buffer contains information that cannot be displayed. Instead, it contains bit-depth information that is used to determine which pixels are visible and which are obscured.

DDSURFACEDESC

```
typedef struct _DDSURFACEDESC{
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwHeight;
    DWORD dwWidth;
    LONG  lPitch;
    DWORD dwBackBufferCount;
    union
    {
        DWORD dwMipMapCount;
        DWORD dwZBufferBitDepth;
        DWORD dwRefreshRate;
    };

    DWORD      dwAlphaBitDepth;
    DWORD      dwReserved;
```

```
    LPVOID     lpSurface;
    DDCOLORKEY ddckCKDestOverlay;
    DDCOLORKEY ddckCKDestBlt;

    DDCOLORKEY    ddckCKSrcOverlay;
    DDCOLORKEY    ddckCKSrcBlt;
    DDPIXELFORMAT ddpfPixelFormat;
    DDSCAPS       ddsCaps;
} DDSURFACEDESC,  FAR* LPDDSURFACEDESC;
```

Contains a description of the surface to be created. This structure is passed to the **IDirectDraw2::CreateSurface** method. The relevant members differ for each potential type of surface.

dwSize
Size of the structure. This member must be initialized before the structure is used.

dwFlags
Optional control flags. One or more of the following flags:

DDSD_ALL
Indicates that all input members are valid.

DDSD_ALPHABITDEPTH
Indicates that the **dwAlphaBitDepth** member is valid.

DDSD_BACKBUFFERCOUNT
Indicates that the **dwBackBufferCount** member is valid.

DDSD_CAPS
Indicates that the **ddsCaps** member is valid.

DDSD_CKDESTBLT
Indicates that the **ddckCKDestBlt** member is valid.

DDSD_CKDESTOVERLAY
Indicates that the **ddckCKDestOverlay** member is valid.

DDSD_CKSRCBLT
Indicates that the **ddckCKSrcBlt** member is valid.

DDSD_CKSRCOVERLAY
Indicates that the **ddckCKSrcOverlay** member is valid.

DDSD_HEIGHT
Indicates that the **dwHeight** member is valid.

DDSD_MIPMAPCOUNT
Indicates that the **dwMipMapCount** member is valid.

DDSD_PITCH
Indicates that the **lPitch** member is valid.

DDSD_PIXELFORMAT

Indicates that the **ddpfPixelFormat** member is valid.

DDSD_REFRESHRATE

Indicates that the **dwRefreshRate** member is valid.

DDSD_WIDTH

Indicates that the **dwWidth** member is valid.

DDSD_ZBUFFERBITDEPTH

Indicates that the **dwZBufferBitDepth** member is valid.

dwHeight
Height of surface.

dwWidth
Width of input surface.

lPitch
Distance to start of next line (return value only).

dwBackBufferCount
Number of back buffers.

dwMipMapCount
Number of mipmap levels.

dwZBufferBitDepth
Depth of z-buffer.

dwRefreshRate
Refresh rate (used when the display mode is described).

dwAlphaBitDepth
Depth of alpha buffer.

dwReserved
Reserved.

lpSurface
Address of the associated surface memory.

ddckCKDestOverlay
Color key for destination overlay use.

ddckCKDestBlt
Color key for destination blit use.

ddckCKSrcOverlay
Color key for source overlay use.

ddckCKSrcBlt
Color key for source blit use.

ddpfPixelFormat
Pixel format description of the surface.

ddsCaps
DirectDraw surface capabilities.

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all methods of the *IDirectDraw2*, *IDirectDrawSurface2*, *IDirectDrawPalette*, and *IDirectDrawClipper* interfaces. For a list of the error codes that each method can return, see the method description.

DD_OK

The request completed successfully.

DDERR_ALREADYINITIALIZED

The object has already been initialized.

DDERR_BLTFASTCANTCLIP

A DirectDrawClipper object is attached to a source surface that has passed into a call to the **IDirectDrawSurface2::BltFast** method.

DDERR_CANNOTATTACHSURFACE

A surface cannot be attached to another requested surface.

DDERR_CANNOTDETACHSURFACE

A surface cannot be detached from another requested surface.

DDERR_CANTCREATEDC

Windows cannot create any more device contexts (DCs).

DDERR_CANTDUPLICATE

Primary and 3D surfaces, or surfaces that are implicitly created, cannot be duplicated.

DDERR_CANTLOCKSURFACE

Access to this surface is refused because an attempt was made to lock the primary surface without DCI support.

DDERR_CANTPAGELOCK

An attempt to page lock a surface failed. Page lock will not work on a display-memory surface or an emulated primary surface.

DDERR_CANTPAGEUNLOCK

An attempt to page unlock a surface failed. Page unlock will not work on a display-memory surface or an emulated primary surface.

DDERR_CLIPPERISUSINGHWND

An attempt was made to set a clip list for a DirectDrawClipper object that is already monitoring a window handle.

DDERR_COLORKEYNOTSET

No source color key is specified for this operation.

DDERR_CURRENTLYNOTAVAIL

No support is currently available.

DDERR_DCALREADYCREATED

A device context (DC) has already been returned for this surface. Only one DC can be retrieved for each surface.

DDERR_DIRECTDRAWALREADYCREATED

A DirectDraw object representing this driver has already been created for this process.

DDERR_EXCEPTION

An exception was encountered while performing the requested operation.

DDERR_EXCLUSIVEMODEALREADYSET

An attempt was made to set the cooperative level when it was already set to exclusive.

DDERR_GENERIC

There is an undefined error condition.

DDERR_HEIGHTALIGN

The height of the provided rectangle is not a multiple of the required alignment.

DDERR_HWNDALREADYSET

The DirectDraw cooperative level window handle has already been set. It cannot be reset while the process has surfaces or palettes created.

DDERR_HWNDSUBCLASSED

DirectDraw is prevented from restoring state because the DirectDraw cooperative level window handle has been subclassed.

DDERR_IMPLICITLYCREATED

The surface cannot be restored because it is an implicitly created surface.

DDERR_INCOMPATIBLEPRIMARY

The primary surface creation request does not match with the existing primary surface.

DDERR_INVALIDCAPS

One or more of the capability bits passed to the callback function are incorrect.

DDERR_INVALIDCLIPLIST

DirectDraw does not support the provided clip list.

DDERR_INVALIDDIRECTDRAWGUID

The globally unique identifier (GUID) passed to the **DirectDrawCreate** function is not a valid DirectDraw driver identifier.

DDERR_INVALIDMODE

DirectDraw does not support the requested mode.

DDERR_INVALIDOBJECT

DirectDraw received a pointer that was an invalid DirectDraw object.

DDERR_INVALIDPARAMS

One or more of the parameters passed to the method are incorrect.

DDERR_INVALIDPIXELFORMAT

The pixel format was invalid as specified.

DDERR_INVALIDPOSITION

The position of the overlay on the destination is no longer legal.

DDERR_INVALIDRECT

The provided rectangle was invalid.

DDERR_INVALIDSURFACETYPE

The requested operation could not be performed because the surface was of the wrong type.

DDERR_LOCKEDSURFACES

One or more surfaces are locked, causing the failure of the requested operation.

DDERR_NO3D

No 3D hardware or emulation is present.

DDERR_NOALPHAHW

No alpha acceleration hardware is present or available, causing the failure of the requested operation.

DDERR_NOBLTHW

No blitter hardware is present.

DDERR_NOCLIPLIST

No clip list is available.

DDERR_NOCLIPPERATTACHED

No DirectDrawClipper object is attached to the surface object.

DDERR_NOCOLORCONVHW

The operation cannot be carried out because no color-conversion hardware is present or available.

DDERR_NOCOLORKEY

The surface does not currently have a color key.

DDERR_NOCOLORKEYHW

The operation cannot be carried out because there is no hardware support for the destination color key.

DDERR_NOCOOPERATIVELEVELSET

A create function is called without the **IDirectDraw2::SetCooperativeLevel** method being called.

DDERR_NODC

No DC has ever been created for this surface.

DDERR_NODDROPSHW

No DirectDraw raster operation (ROP) hardware is available.

DDERR_NODIRECTDRAWHW

Hardware-only DirectDraw object creation is not possible; the driver does not support any hardware.

DDERR_NODIRECTDRAWSUPPORT

DirectDraw support is not possible with the current display driver.

DDERR_NOEMULATION

Software emulation is not available.

DDERR_NOEXCLUSIVEMODE

The operation requires the application to have exclusive mode, but the application does not have exclusive mode.

DDERR_NOFLIPHW

Flipping visible surfaces is not supported.

DDERR_NOGDI

No GDI is present.

DDERR_NOHWND

Clipper notification requires a window handle, or no window handle has been previously set as the cooperative level window handle.

DDERR_NOMIPMAPHW

The operation cannot be carried out because no mipmap texture mapping hardware is present or available.

DDERR_NOMIRRORHW

The operation cannot be carried out because no mirroring hardware is present or available.

DDERR_NOOVERLAYDEST

The **IDirectDrawSurface2::GetOverlayPosition** method is called on an overlay that the **IDirectDrawSurface2::UpdateOverlay** method has not been called on to establish a destination.

DDERR_NOOVERLAYHW

The operation cannot be carried out because no overlay hardware is present or available.

DDERR_NOPALETTEATTACHED

No palette object is attached to this surface.

DDERR_NOPALETTEHW

There is no hardware support for 16- or 256-color palettes.

DDERR_NORASTEROPHW

The operation cannot be carried out because no appropriate raster operation hardware is present or available.

DDERR_NOROTATIONHW

The operation cannot be carried out because no rotation hardware is present or available.

DDERR_NOSTRETCHHW

The operation cannot be carried out because there is no hardware support for stretching.

DDERR_NOT4BITCOLOR

The DirectDrawSurface object is not using a 4-bit color palette and the requested operation requires a 4-bit color palette.

DDERR_NOT4BITCOLORINDEX

The DirectDrawSurface object is not using a 4-bit color index palette and the requested operation requires a 4-bit color index palette.

DDERR_NOT8BITCOLOR

The DirectDrawSurface object is not using an 8-bit color palette and the requested operation requires an 8-bit color palette.

DDERR_NOTAOVERLAYSURFACE

An overlay component is called for a non-overlay surface.

DDERR_NOTEXTUREHW

The operation cannot be carried out because no texture-mapping hardware is present or available.

DDERR_NOTFLIPPABLE

An attempt has been made to flip a surface that cannot be flipped.

DDERR_NOTFOUND

The requested item was not found.

DDERR_NOTINITIALIZED

An attempt was made to call an interface method of a DirectDraw object created by **CoCreateInstance** before the object was initialized.

DDERR_NOTLOCKED

An attempt is made to unlock a surface that was not locked.

DDERR_NOTPAGELOCKED

An attempt is made to page unlock a surface with no outstanding page locks.

DDERR_NOTPALETTIZED

The surface being used is not a palette-based surface.

DDERR_NOVSYNCHW

The operation cannot be carried out because there is no hardware support for vertical blank synchronized operations.

DDERR_NOZBUFFERHW

The operation to create a z-buffer in display memory or to perform a blit using a z-buffer cannot be carried out because there is no hardware support for z-buffers.

DDERR_NOZOVERLAYHW

The overlay surfaces cannot be z-layered based on the z-order because the hardware does not support z-ordering of overlays.

DDERR_OUTOFCAPS

The hardware needed for the requested operation has already been allocated.

DDERR_OUTOFMEMORY

DirectDraw does not have enough memory to perform the operation.

DDERR_OUTOFVIDEOMEMORY

DirectDraw does not have enough display memory to perform the operation.

DDERR_OVERLAYCANTCLIP

The hardware does not support clipped overlays.

DDERR_OVERLAYCOLORKEYONLYONEACTIVE

An attempt was made to have more than one color key active on an overlay.

DDERR_OVERLAYNOTVISIBLE

The **IDirectDrawSurface2::GetOverlayPosition** method is called on a hidden overlay.

DDERR_PALETTEBUSY

Access to this palette is refused because the palette is locked by another thread.

DDERR_PRIMARYSURFACEALREADYEXISTS

This process has already created a primary surface.

DDERR_REGIONTOOSMALL

The region passed to the **IDirectDrawClipper::GetClipList** method is too small.

DDERR_SURFACEALREADYATTACHED

An attempt was made to attach a surface to another surface to which it is already attached.

DDERR_SURFACEALREADYDEPENDENT

An attempt was made to make a surface a dependency of another surface to which it is already dependent.

DDERR_SURFACEBUSY

Access to the surface is refused because the surface is locked by another thread.

DDERR_SURFACEISOBSCURED

Access to the surface is refused because the surface is obscured.

DDERR_SURFACELOST

Access to the surface is refused because the surface memory is gone. The DirectDrawSurface object representing this surface should have the **IDirectDrawSurface2::Restore** method called on it.

DDERR_SURFACENOTATTACHED

The requested surface is not attached.

DDERR_TOOBIGHEIGHT

The height requested by DirectDraw is too large.

DDERR_TOOBIGSIZE

The size requested by DirectDraw is too large. However, the individual height and width are OK.

DDERR_TOOBIGWIDTH

The width requested by DirectDraw is too large.

DDERR_UNSUPPORTED

The operation is not supported.

DDERR_UNSUPPORTEDFORMAT

The FourCC format requested is not supported by DirectDraw.

DDERR_UNSUPPORTEDMASK

The bitmask in the pixel format requested is not supported by DirectDraw.

DDERR_UNSUPPORTEDMODE

The display is currently in an unsupported mode.

DDERR_VERTICALBLANKINPROGRESS

A vertical blank is in progress.

DDERR_WASSTILLDRAWING

The previous blit operation that is transferring information to or from this surface is incomplete.

DDERR_WRONGMODE

This surface cannot be restored because it was created in a different mode.

DDERR_XALIGN

The provided rectangle was not horizontally aligned on a required boundary.