Copyrights and Trademarks

Using DirectX 2 in Windows

The Microsoft® DirectX™ 2 Software Development Kit (SDK) provides a finely-tuned set of application programming interfaces (APIs) that give you, as a developer, the resources needed to design high-performance, real-time applications, such as the next generation of computer games and multimedia applications.

Microsoft developed the DirectX 2 SDK for a number of reasons. The primary reason is to make performance on Windows-based platforms rival or exceed performance on MS-DOS-based platforms and game console-system platforms. Other reasons are to promote game development for Microsoft Windows®, and to assist you by providing a robust, standardized, and well-documented platform for which to write games.

Reasons for Developing DirectX Windows Applications

DirectX was developed to provide Microsoft Windows applications with high-performance, real-time access to available hardware on current computer systems. DirectX provides a consistent interface between hardware manufacturers and you, the application developer, thereby reducing the complexity of installation and configuration while utilizing the hardware to its best advantage.

One of the primary reasons for creating DirectX was to promote games development on the Windows platform. The majority of games developed for the personal computer today are MS-DOS-based. However, when developing MS-DOS-based games, you must conform to a number of hardware implementations for a variety of cards, which complicates installation. In addition, development of MS-DOS-based games can be much more complex on a personal computer than on a console system, due to the generalized processor, greater RAM size, and persistent storage of the personal computer.

A high-performance Windows-based game will:

- Install successfully.
- Take advantage of hardware accelerator cards designed specifically for improving performance.
- Take advantage of Windows hardware and software standards such as Plug and Play.
- Take advantage of the communications services built into Windows.

Providing Standards for Hardware Accelerators

The primary goals of the DirectX 2 SDK are to provide portable access to the features in use with MS-DOS® today, to not compromise MS-DOS or game console performance, and to remove the obstacles to hardware innovation on the personal computer platform.

Another important goal is to provide guidelines for hardware companies based on feedback from high-performance application developers and independent hardware vendors (IHVs). Therefore, the DirectX 2 SDK components often provide specifications for hardware accelerator features that do not yet exist. In many cases, these specifications are emulated in the software. In other cases, the capabilities of the hardware must be polled first, and the feature bypassed if it is not supported.

Some of the display hardware features coming out in the near future include:

- Overlays. Overlays will be supported so page flipping will be enabled within a window in a graphic device interface (GDI). *Page flipping* is the double-buffer scheme used to display frames on the entire screen.
- Sprite engines, used to make overlaying sprites easier.
- Stretching with interpolation. Stretching a smaller frame to fit the entire screen can be an efficient way to conserve display RAM.
- Alpha blending, used to mix colors at the hardware pixel level.
- Three-dimensional (3D) accelerators with perspective-correct textures. This allows textures to be displayed on a 3D surface; for example, hallways in a castle generated by 3D software can be textured with a brick wall bitmap that maintains the correct perspective.
- Z-buffer-aware blits for 3D graphics.
- 2 megabytes (MB) of display memory as standard. 3D games, for example, generally need at least this much display RAM.
- A compression standard so you can put more data into display memory. This standard will be used for textures, will include transparency compression, and will be very fast when implemented in software as well as hardware.

Audio hardware features that will be released include:

- 3D audio enhancers that provide a spatial placement for different sounds. This will be particularly effective with headphones.
- Onboard memory for audio boards.
- Audio-video combination boards that share onboard memory.

In addition, video playback will see the benefit of hardware accelerators that will be compatible with the DirectX 2 SDK. One of the features that will be supported by future releases of the DirectX 2 SDK is hardware-accelerated decompression of YUV video.

DirectX 2 Components

The DirectX 2 SDK is composed of several interfaces that address and answer the performance issues of programming games and high-performance applications in the Windows 95 operating system:

- The Microsoft DirectDraw™ application programming interface. This accelerates hardware and software animation techniques by providing direct access to bitmaps in off-screen display memory as well as extremely fast access to the blitting and buffer-flipping capabilities of the hardware.
- The Microsoft DirectSound™ application programming interface. This enables hardware and software sound mixing and playback.
- The Microsoft DirectPlay™ application programming interface. This allows easy connectivity of games over a modem link or network.
- The Microsoft Direct3D™ application programming interface. This provides a high-level Retained-mode interface that allows applications to easily implement a complete 3D graphical system, and a low-level Immediate-mode interface that applications can use to take complete control over the rendering pipeline.
- The Microsoft DirectInput™ application programming interface. This provides joystick input capabilities to your game that are scalable to future Windows hardware input APIs and drivers.
- The Microsoft AutoPlay feature of the Microsoft Windows 95 operating system. This lets your CD run an installation program or the game itself immediately upon insertion of the CD.

The last two features, DirectInput and AutoPlay, exist in Microsoft Win32® application programming interface and are not unique to the DirectX 2 SDK.

DirectDraw

The biggest gain in performance in the DirectX 2 SDK comes from the DirectDraw services, a combination of four Component Object Model (COM) interfaces: IDirectDraw, IDirectDrawSurface, IDirectDrawPalette, and IDirectDrawClipper. For more information about the COM concepts required for programming applications using the DirectX 2 SDK, see The Component Object Model.

A DirectDraw object, created using the DirectDrawCreate function, represents the display adapter card. One of the object's methods, IDirectDraw::CreateSurface, creates 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 surface is created in addition to the primary surface and exchanges 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 being composed. This is done by transferring a series of off-screen bitmaps stored on other DirectDrawSurface objects in display RAM. The IDirectDrawSurface::Flip method is called to display the recently composed frame, which sets a register so the exchange occurs when the screen performs a vertical retrace. This operation is asynchronous, so the application can continue processing after calling **IDirectDrawSurface::Flip**. (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 **IDirectDrawSurface::Flip**, and so on.

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

The third type of 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 only available when a game has exclusive access to the display hardware. DirectDrawPalette objects are also created from DirectDraw objects.

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

Transparent blitting is the technique by which 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 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.

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, plus extra capabilities associated only with overlays. These capabilities require extra display memory, and 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.

DirectSound

Programming for high-performance applications and games requires efficient and dynamic sound production. Microsoft provides two methods for achieving this: MIDI streams and DirectSound. MIDI streams are actually part of the Windows 95 multimedia application programming interface. They provide the ability to time stamp MIDI messages and send a buffer of these messages to the system, which can then efficiently integrate them with its processes. For more information about MIDI streams, see the documentation included with the Win32 SDK.

DirectSound implements a new model for playing back digitally-recorded sound samples and mixing different sample sources together. As with other object classes in the DirectX 2 SDK, DirectSound uses the hardware to its greatest advantage whenever possible and emulates hardware features in software when the feature is not present in hardware. You can query hardware capabilities at run time to determine the best solution for any given personal computer configuration.

DirectSound is built on the COM-based interfaces IDirectSound and IDirectSoundBuffer, and is extensible to other interfaces. For more information about the COM concepts required for programming applications using the DirectX 2 SDK, see The Component Object Model.

The DirectSound object represents the sound card and its various attributes. The DirectSoundBuffer object is created using the DirectSound object's IDirectSound::CreateSoundBuffer method and represents a buffer containing sound data. Several DirectSoundBuffer objects can exist and be mixed together into the primary DirectSoundBuffer object. DirectSound buffers are used to start, stop, and pause sound playback, and to set attributes such as frequency, format, and so on.

Depending on the card type, DirectSound buffers can exist in hardware as onboard RAM, wave table memory, a direct memory access (DMA) channel, or a virtual buffer (for an I/O port-based audio card). Where there is no hardware implementation of a DirectSound buffer, it is emulated in system memory.

The primary buffer is generally used to mix sound from secondary buffers, but can be accessed directly for custom mixing or other specialized activities. (Use caution in locking the primary buffer, because this blocks all access to the sound hardware from other sources).

The secondary buffers can store common sounds played throughout an application, such as in a game. A sound stored in a secondary buffer can be played as a single event or as a looping sound that plays repeatedly.

Secondary buffers can also play sounds larger than available sound buffer memory. When used to play a sound that is larger than the buffer, the secondary buffer serves as a queue that stores the portions of the sound about to be played.

DirectPlay

One of the most compelling features of the personal computer as a game platform is its easy access to communication services. DirectPlay is a service that capitalizes on this capability and allows multiple players to interact with a game through standard modems, network connections, or online services.

The IDirectPlay interface contains methods providing capabilities such as creating and destroying players, adding players to and deleting players from groups, sending messages to players, inviting players to participate in a game, and so on.

DirectPlay is composed of the interface to the game, as defined by the IDirectPlay interface and the DirectPlay server. DirectPlay servers are provided by Microsoft for modems and networks, as well as by third parties. When using a supported server, DirectPlay-enabled games can bypass connectivity and communication overhead details.

Direct3D

Direct3D is the next generation of real-time, interactive 3D technology for mainstream computer users on the desktop and the Internet. It provides the API services and device independence that you need, delivers a common driver model for hardware vendors, enables turn-key 3D solutions to be offered by personal computer manufacturers, and makes it easy for end-users to add high-end 3D to their systems.

Direct3D is a complete set of real-time 3D graphics services that delivers fast software-based rendering of the full 3D rendering pipeline (transformations, lighting, and rasterization) and transparent access to hardware acceleration. Direct3D offers a comprehensive next-generation 3D solution for mainstream computers. API services include an integrated high-level Retained mode and low-level Immediate mode API, and support for other systems that might use Direct3D to gain access to 3D hardware acceleration. Direct3D is fully scalable, enabling all or part of the 3D rendering pipeline to be accelerated by hardware. Direct3D exposes advanced graphics capabilities of 3D hardware accelerators, including z-buffering, anti-aliasing, alpha blending, mipmapping, atmospheric effects, and perspective-correct texture mapping. Tight integration with other DirectX technologies enables Direct3D to deliver such advanced features as video mapping, hardware 3D rendering in 2D overlay planes—and even sprites—providing seamless use of 2D and 3D graphics in interactive media titles.

DirectInput

The joystick represents a class of devices that report tactile movements and actions players make within a game. DirectInput provides the functionality to process the data representing these movements and actions from joysticks, as well as other related devices, such as trackballs and flight harnesses.

Currently, DirectInput is simply another name for an existing Win32 function, **joyGetPosEx**. This function provides extended capabilities to its predecessor, **joyGetPos**, and should be used for any joystick services. In future support for input devices, including virtual reality hardware, games that use **joyGetPosEx** will be automatically supported for joystick input services. This is not the case for **joyGetPos**.

AutoPlay

AutoPlay is a feature of Windows 95 that automatically plays a CD-ROM or audio CD when inserted into a CD-ROM drive. While this feature is not specific to the Windows 95 DirectX 2 SDK; any CD-ROM product that bears the Windows 95 logo must be enabled with the AutoPlay feature.

Sample Applications

Sample applications that demonstrate the components of the Windows 95 DirectX 2 SDK are located in the SDK\SAMPLES directory.

The Component Object Model

Many of the APIs in the DirectX 2 SDK are composed of objects and interfaces based on the Component Object Model (COM). COM is a foundation for an object-based system that focuses on reuse of interfaces and is the model at the heart of OLE programming. It is also an interface specification from which any number of interfaces can be built. It is an object model at the operating system level.

Many DirectX 2 APIs are instantiated as a set of OLE objects. The *object* can be considered a black box that represents the hardware and requires communication with applications through an *interface*. The commands sent to and from the object through the COM interface are called *methods*. For example, the IDirectDraw::GetDisplayMode method is sent through the IDirectDraw interface to get the current display mode of the display adapter from the DirectDraw object.

Objects can bind to other objects at run time and use the implementation of interfaces provided by the other object. If you know an object is an OLE object, and what interfaces that object supports, your application, or another object, can determine what services the first object can be called upon to perform. One of the methods inherited by all OLE objects, called **QueryInterface**, lets you determine what interfaces are supported by an object and create pointers to these interfaces.

IUnknown

All COM interfaces are derived from an interface called IUnknown. The IUnknown interface provides DirectX with control of the object's lifetime, and the ability to navigate multiple interfaces. IUnknown has only three methods:

- **AddRef**, which increments the reference count of the object by 1 when an interface or another application binds itself to the object.
- **Release**, which decrements the reference count of the object by 1. When the count reaches 0, the object is deallocated.
- **QueryInterface**, which queries the object about the features it supports by asking for pointers to a specific interface.

AddRef and **Release** maintain the reference count of a particular object. For example, if you create a DirectDrawSurface object, the reference count of the object is set to 1. Each time a function returns a pointer to an interface for that object, the function must then call **AddRef** through that pointer to increment the reference count. All **AddRef** calls must be matched with a call to **Release**. Before the pointer can be destroyed, you must call **Release** through that pointer. Once the reference count of a particular object reaches 0, the object is destroyed and all interfaces to that object are then invalid.

QueryInterface determines whether an object supports a specific interface. If the interface is supported, **QueryInterface** returns a pointer to that particular interface. You can then use the methods contained in that interface to communicate with the object. If **QueryInterface** successfully returns a pointer to an interface, it implicitly calls **AddRef** to increment the reference count, so your application must call **Release** to decrement the reference count before destroying the pointer to the interface.

DirectX 2 SDK COM Interfaces

The interfaces in the DirectX 2 SDK have been created at a very basic level of the COM programming hierarchy. Each main device object interface, such as IDirectDraw, IDirectSound, or IDirectPlay, derives directly from the IUnknown interface in OLE. Creation of these basic objects is handled by specialized functions in the dynamic link library (DLL) for each object rather than by the Win32 **CoCreateInstance** function typically used to create COM objects.

In general, the DirectX 2 SDK object model provides one main object for each device, from which other support service objects are derived. For example, the DirectDraw object represents the display adapter. It is used to create DirectDrawSurface objects that represent the display RAM and DirectDrawPalette objects that represent hardware palettes. Similarly, the DirectSound object represents the audio card and creates DirectSoundBuffer objects that represent the sound sources on that card.

Besides the ability to generate subordinate objects, the main device object determines the capabilities of the hardware device it represents, such as the screen size and number of colors, or whether the audio card has wave table synthesis.

C++ and the COM Interface

To C++ programmers, a COM interface is like an abstract base class. That is, it defines a set of signatures and semantics but not the implementation, and no state data is associated with the interface. In a C++ abstract base class, all methods are defined as "pure virtual," which means they have no code associated with them.

Pure virtual C++ functions and COM interfaces both employ a device called a *vtable*. A vtable contains the addresses of all functions that implement the given interface. If you want a program or object to use these functions, you can use the **QueryInterface** method to verify the interface exists on an object, and to obtain a pointer to that interface. What your program or object actually receives from the object after sending **QueryInterface** is a pointer to the vtable, through which this method can call the interface methods implemented by the object. This mechanism totally isolates private data used by the object and the calling client process.

Another similarity of COM objects to C++ objects is that the first argument of a method is the name of the interface or class, called the *this* argument in C++. Because COM objects and C++ objects are totally binary compatible, the compiler treats COM interfaces like C++ abstract classes and assumes the same syntax. This results in less complex code. For example, the *this* argument in C++ is treated as an understood parameter and not coded, and the indirection through the vtable is handled implicitly in C++.

Accessing COM Objects Using C

Any COM interface method can be called from a C program. There are two things you need to remember when calling an interface method from C:

- The first parameter of the method is always a reference to the object that has been created and is invoking the method (the *this* argument).
- Each method in the interface is referenced through a pointer to the object's vtable.

The following example creates a surface associated with a DirectDraw object by calling the IDirectDraw::CreateSurface method using the C programming language:

```
ret = lpDD->lpVtbl->CreateSurface (lpDD, &ddsd, &lpDDS,
      NULL);
```

The DirectDraw object associated with the new surface is referenced by the *lpDD* parameter. Incidentally, this method fills in a surface description structure (*&ddsd*) and returns a pointer the new surface (*&lpDDS*).

To call the **IDirectDraw::CreateSurface** method, you first dereference the DirectDraw object's vtable, then dereference the method from the vtable. The first parameter supplied in the method is a reference to the DirectDraw object that has been created and is invoking the method.

To illustrate the difference between calling a COM object method in C and C++, the same method in C++ is shown below (C++ implicitly dereferences the vtbl pointer and passes the *this* pointer):

```
ret = lpDD->CreateSurface(&ddsd, &lpDDS, NULL)
```

Interface Method Names and Syntax

All of the COM interface methods described in this document are shown using C++ class names. This naming convention is used for consistency, and to differentiate between methods used for different DirectX objects that use the same name, such as **QueryInterface**, **AddRef**, and **Release**. This does not imply that these methods can only be used with C++.

In addition, the syntax provided for the methods uses C++ conventions for consistency. It does not include the *this* pointer to the interface. When programming in C, the pointer to the interface must be included in each method. For example, the following example shows the C++ syntax for the IDirectDraw::GetCaps method:

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

The same example using C syntax looks like this:

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

The *lpDD* parameter is a pointer to the DirectDraw structure that represents the DirectDraw object.

Using Macro Definitions

Many of the header files for the DirectX interfaces include macro definitions for each method. These macros are included to simplify the use of the methods in your programming.

For example, the following example uses the IDirectDraw_CreateSurface macro to call the IDirectDraw::CreateSurface method. The first parameter is a reference to the DirectDraw object that has been created and is invoking the method:

```
ret = IDirectDraw_CreateSurface (lpDD, &ddsd, &lpDDS,
    NULL);
```

To obtain a current list of the methods supported by macro definitions, see the appropriate header file for the DirectX component you want to use.

Floating-point Precision

The DirectX architecture uses a floating-point precision of 53. If your application needs to change this precision, it must be changed back to 53 when the calculations are finished. Otherwise, system components that depend on the default value will stop working.

Differences Between the Game SDK and the DirectX 2 SDK

The DirectX 2 SDK provides more services—and more avenues for innovation—than did the Game SDK. Although the DirectX 2 SDK contains additional functions and services, all of the applications you have written with the Game SDK will compile and run successfully without changes.

This section identifies some of the most significant differences and improvements of the DirectX 2 SDK over the Game SDK. The purpose of this section is to help developers familiar with the Game SDK quickly identify several important areas of the DirectX SDK that are significantly different.

DirectDraw

The DirectDraw API functions have been significantly improved over those found in the Game SDK. The following list briefly describes the major improvements:

- The IDirectDraw2 and IDirectDrawSurface2 interfaces were added. For more information, see IDirectDraw2 Interface and IDirectDrawSurface2 Interface.
- The following flags were added:

  DDBLT_DEPTHFILL
  DDCAPS_BLTDEPTHFILL
  DDCAPS_CANBLTSYSMEM
  DDCAPS_CANCLIP
  DDCAPS_CANCLIPSTRETCHED
  DDCAPS2_NO2DDURING3DSCENE
  DDEDM_REFRESHRATES
  DDPCAPS_1BIT
  DDPCAPS_2BIT
  DDPF_PALETTEINDEXED1
  DDPF_PALETTEINDEXED2
  DDSCAPS_ALLOCONLOAD
  DDSCAPS_MIPMAP
  DDSD_MIPMAPCOUNT
  DDSD_REFRESHRATE

  In addition, the name of the DDSCAPS_TEXTUREMAP flag was changed to DDSCAPS_TEXTURE and the name of the DDPF_PALETTEINDEXED4TO8 flag was changed to DDPF_PALETTEINDEXEDTO8.
- The following error messages were added:

  DDERR_CANTLOCKSURFACE
  DDERR_CANTPAGELOCK
  DDERR_CANTPAGEUNLOCK
  DDERR_DCALREADYCREATED
  DDERR_INVALIDSURFACETYPE
  DDERR_NOMIPMAPHW
  DDERR_NOTPAGELOCKED
  DDERR_NOTINITIALIZED

- The IDirectDraw2::SetDisplayMode method contains two new parameters, *dwRefreshRate* and *dwFlags*. If neither of these parameters are needed, you can still use **IDirectDraw::SetDisplayMode**.
- The IDirectDraw2::EnumDisplayModes method was added to enumerate the refresh rate of the monitor and store it in the **dwRefreshRate** member of the DDSURFACEDESC structure.
- A new method has been added to the IDirectDraw2 interface: IDirectDraw2::GetAvailableVidMem.
- Three new methods have been added to the IDirectDrawSurface2 interface: IDirectDrawSurface2::GetDDInterface, IDirectDrawSurface2::PageLock, and IDirectDrawSurface2::PageUnlock.
- DirectDraw in the Game SDK limited the available display modes to 640 by 480 with pixel depths of 8 bpp and 16 bpp. DirectDraw now allows an application to change the mode to allow any supported by the display driver.
- DirectDraw now checks the display modes it is capable of using against the display restrictions of the installed monitor. If the requested mode is not compatible with the monitor, then the **IDirectDraw2::SetDisplayMode** method will fail. Only modes supported on the installed monitor will be enumerated in the **IDirectDraw2::EnumDisplayModes** method.
- In the Game SDK, DirectDraw only allowed the creation of one DirectDraw object per process. If your process happened to use another system component, such as DirectPlay, that created a DirectDraw object, the process would be unable to create another DirectDraw object for its own use. It is now possible for a process to call the DirectDrawCreate function as many times as necessary. A unique and independent interface will be returned from each call. For more information, see Multiple DirectDraw Objects per Process.
- DirectDraw on the Game SDK required an HWND be specified in the IDirectDraw::SetCooperativeLevel method call regardless of whether or not a full-screen, exclusive mode was being requested. This method no longer requires an HWND be specified if the application is requesting DDSCL_NORMAL mode. It is now possible for an application to use DirectDraw with multiple windows. All of these windows can be used simultaneously in normal mode.
- In DirectDraw on the Game SDK, if a surface was in system memory, the hardware emulation layer (HEL) automatically performed the blit. However, some display cards have DMA hardware that allows them to efficiently blit to and from system memory surfaces. In the DirectX 2 version of

DirectDraw, the DDCAPS structure has been expanded to allow drivers to report this capability. The following members have been added:

```
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       dwSSBFXCaps
DWORD       dwSSBRops[DD_ROP_SPACE]
```

- In DirectDraw on the Game SDK, palettes could only be attached to the primary surface. Palettes can now be attached to any paletized surface (primary, back buffer, off-screen plain, or texture map). For more information, see Setting Palettes on Non-Primary Surfaces.
- Palettes can now be shared between multiple surfaces. For more information, see Sharing Palettes.
- In DirectDraw on the Game SDK, only 8-bit (256 entry) palettes were supported. DirectDraw on the DirectX 2 SDK supports 1-bit (2 entry), 2-bit (4 entry), and 4-bit (16 entry) palettes as well. For more information, see New Palette Types.
- Clippers can now be shared between multiple surfaces. For more information, see Sharing Clippers.
- A new API function, DirectDrawCreateClipper was added. This function allows clipper objects to be created that are not owned by a DirectDraw object. For more information, see Driver Independent Clippers.
- In DirectDraw on the Game SDK, the HEL could only create surfaces whose pixel format exactly matched that of the current primary surface. This restriction has been relaxed for the DirectX 2 version of DirectDraw. For more information, see **Surface Format Support in the Hardware Emulation Layer (HEL)**.
- Support for surfaces specific to 3D rendering (texture maps, mipmaps, and z-buffers) has been added or enhanced in the DirectX 2 version of DirectDraw. For more information, see Texture Maps, Mipmaps, and Z-Buffers.

DirectSound

Although there are few visible external differences between DirectSound on the Game SDK and DirectSound on the DirectX 2 SDK, significant internal improvements have been made. The following is a brief list of some of these differences:

- Improvements to the wave emulation code, used when no device driver is available, to support wave drivers that do not work correctly.
- Changes to the sound focus management. These changes should not make any difference to most games. They were needed to support out-of-proc (exe server) COM objects, and to support the requirements of ActiveX™ (Microsoft's high-level media streaming architecture).
- The DSBCAPS_STICKYFOCUS flag was added. This flag can be specified in a IDirectSound::CreateSoundBuffer method call in order to change the focus behavior of the sound buffer.

DirectPlay

The difference below reflects the only external change that has been made to DirectPlay:

- The error DPERR_SENDTOOBIG is now returned if the message buffer passed to the IDirectPlay::Send method is larger than DirectPlay allows.

Direct3D

The Game SDK did not contain any Direct3D functionality. All of the Direct3D functionality is new in the DirectX 2 SDK.

DirectInput

No changes were made to this version of DirectInput.

AutoPlay

No changes were made to this version of AutoPlay.

DirectSetup

The DSETUP_D3D flag was added to the *dwFlags* parameter of the DirectXSetup function.

Conventions

The following conventions are used to define syntax.

Convention	Meaning
Italic text	Denotes a placeholder or variable. You must provide the actual value. For example, the statement SetCursorPos(*X*, *Y*) requires you to substitute values for the *X* and *Y* parameters.
[]	Enclose optional parameters.
|	Separates an either/or choice.
...	Specifies that the preceding item may be repeated.
.	Represents an omitted portion of a sample application.

.

.

In addition, certain typographic conventions are used to help you understand this material.

Convention	Meaning
SMALL CAPITALS	Indicates the names of keys, key sequences, and key combinations—for example, ALT+SPACEBAR.
FULL CAPITALS	Indicates most type and structure names, which are also bold, and constants.
`monospace`	Sets off code examples and shows syntax spacing.

About DirectDraw

DirectDraw™ is a DirectX™ 2 SDK component that allows direct manipulation of display memory, hardware blitters, hardware overlays, and page flipping. DirectDraw provides this functionality while maintaining compatibility with existing Microsoft® Windows® 95 applications and device drivers.

The DirectX 2 SDK allows you an unprecedented level of access to display and audio hardware, while it insulates you from the specific details of that hardware. DirectX 2 is built for speed. In keeping with these goals, DirectDraw is not a high-level graphics application programming interface (API).

DirectDraw for Windows 95 is a software interface that provides direct access to display devices while maintaining compatibility with the Windows graphics device interface (GDI). 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 display device-dependent features.

DirectDraw works with a wide variety of display hardware, ranging from simple SVGAs to advanced hardware implementations that provide clipping, stretching, and non-RGB color format support. The interface is designed so that your applications can request the capabilities of the underlying hardware, then use those capabilities as required.

DirectDraw provides the following display device-dependent benefits:

- Supports double-buffered and page flipping graphics.
- Provides access to, and control of, the display card's blitter.
- Supports 3D z-buffers.
- Supports hardware assisted overlays with z-ordering.
- Improves graphics quality by providing access to image-stretching hardware.
- Provides 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. Your application need only recognize some basic device dependencies that are standard across hardware implementations, such as RGB and YUV color formats and the stride between raster lines. You need not worry about the specific calling procedures required to utilize the blitter or manipulate palette registers. Essentially, DirectDraw manages display memory. 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®.

Introduction to DirectDraw

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

DirectDraw is the Windows system component that performs the common functions required by both hardware and software implementations of DirectDraw. As such, DirectDraw is the only client of the DirectDraw hardware abstraction layer (HAL). Applications must write to DirectDraw. DirectDraw returns 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 and adjust itself accordingly.

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 hWnds to provide clip lists, and all other device-independent functionality.

DirectDraw HAL

The DirectDraw 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 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 should not report it as a hardware capability. The HAL should do no parameter validation. The parameters will be validated by DirectDraw before the HAL is invoked.

DirectDraw Software Emulation

DirectDraw's hardware emulation layer (HEL) presents its capabilities to DirectDraw just like 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. There can be one DirectDraw object for every logical display device in operation. An application development environment, for instance, might have two monitors, one running the application using DirectDraw and one running the development environment using GDI.

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 can be created in system memory if required by the hardware design or if DirectDraw is doing software emulation. An overlay is a surface that can be made visible without altering the pixels it is obscuring. Overlays and sprites are synonymous. A texture map is a surface that can be wrapped onto a 3D surface.

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 may refuse the request to attach a DirectDrawPalette to it. This is not unusual because most hardware does not support multiple palettes.

IDirectDraw2 Interface

The COM model used by DirectDraw specifies that new functionality can be added by providing new interfaces. This release of DirectDraw implements two new interfaces, the **IDirectDraw2 Interface** and the IDirectDrawSurface2 Interface. These new interfaces 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 a compile-time error.

In addition to the **IDirectDraw2::GetAvailableVidMem** method, the IDirectDraw2 interface contains all of the methods provided in the IDirectDraw interface. Included in this interface are the IDirectDraw2::SetDisplayMode and IDirectDraw2::EnumDisplayModes methods which allow refresh rates to be specified. If refresh rates are not required, the **IDirectDraw::SetDisplayMode** and **IDirectDraw::EnumDisplayModes** methods can be used.

The interaction between the IDirectDraw::SetCooperativeLevel and **IDirectDraw::SetDisplayMode** methods is slightly different than the one between the **IDirectDraw2::SetCooperativeLevel** and IDirectDraw2::SetDisplayMode methods. If you are using the IDirectDraw interface and an application gains exclusive mode by calling **IDirectDraw::SetCooperativeLevel** with the DDSCL_EXCLUSIVE flag, changes the mode using **IDirectDraw::SetDisplayMode**, 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.

Note Because some methods behave somewhat differently in the two interfaces, mixing methods from IDirectDraw and IDirectDraw2 may cause unpredictable results. You should only use functions from one of these interfaces at a time; do not use some functions from IDirectDraw and other functions from IDirectDraw2.

For more information on using the IDirectDrawSurface2 interface, see IDirectDrawSurface2 Interface.

Multiple DirectDraw Objects per Process

DirectDraw allows a process to call the DirectDrawCreate function as many times as necessary. A unique and independent interface will be returned from 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 two different processes.

Since 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 with 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 hardware emulation layer (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 95 allows a user to specify the type of monitor that is being used. DirectDraw checks the display modes that it knows about against the display restrictions of the installed monitor. If it is determined that the requested mode is not compatible with the monitor, the IDirectDraw2::SetDisplayMode method call will fail. Only modes that are supported on the installed monitor will be enumerated with 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 mode is changed, all of the surfaces stored in display memory are lost (for more information, see Losing Surfaces).

DirectDraw implicitly creates a GDIPrimarySurface when it is instantiated for a display device 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 that owns GDI's primary surface can always be obtained using the IDirectDraw::GetGDISurface method.

GDI is not allowed to 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

Display modes can be changed using the IDirectDraw2::SetDisplayMode method. Modes can be changed by any application as long as all of the applications are sharing the display card.

The DirectDraw exclusive mode does not bar other applications from allocating DirecDrawSurfaces, nor does it exclude them from using DirectDraw or GDI functionality. However, it does prevent all applications, other than the one that obtained exclusive access, from changing the display mode or changing the palette.

Creating DirectDraw Objects Using CoCreateInstance

You can create a DirectDraw object using **CoCreateInstance** and the IDirectDraw::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 using CoInitialize(NULL).

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

2 Create your DirectDraw object using **CoCreateInstance** and the **IDirectDraw::Initialize** method.

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

CLSID_DirectDraw is the class identifier of the DirectDraw driver object class and *IID_IDirectDraw* is the particular DirectDraw interface you want. *lpdd* is the DirectDraw object returned. **CoCreateInstance** returns an uninitialized object.

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

Before closing the application, shut down COM using **CoUninitialize**, as shown below.

```
    CoUnitialize();
```

IDirectDrawSurface2 Interface

The COM model that DirectDraw uses specifies that new functionality can be added by providing new interfaces. This release of DirectDraw implements two new interfaces, the IDirectDraw2 Interface and the **IDirectDrawSurface2 Interface**.

The following example shows how to create an **IDirectDrawSurface2** interface:

```
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, along with three new methods: IDirectDrawSurface2::GetDDInterface, IDirectDrawSurface2::PageLock, and IDirectDrawSurface2::PageUnlock.

For more information on obtaining the IDirectDraw2 interface, see IDirectDraw2 Interface.

Frame Buffer Access

DirectDrawSurface objects represent surface memory in the DirectDraw architecture. A DirectDrawSurface allows an application to directly gain access to this surface memory through the IDirectDrawSurface::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 **IDirectDrawSurface::Lock** with a NULL RECT, it is assumed that exclusive access to the entire piece of surface memory is being requested by the application. 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 with the IDirectDrawSurface::Unlock method.

Experience shows that there are several common problems you may encounter when rendering directly into a DirectDrawSurface. The following list describes some solutions to these problems:

- Never assume a constant display pitch. Always examine the pitch information returned by the **IDirectDrawSurface::Lock** method. This pitch may 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 being used.
- Limit activity between the calls to the **IDirectDrawSurface::Lock** and **IDirectDrawSurface::Unlock** methods. The **IDirectDrawSurface::Lock** method holds the WIN16 lock so that gaining access to surface memory can occur safely, and the IDirectDrawSurface::GetDC method implicitly calls **IDirectDrawSurface::Lock**. The WIN16 lock serializes access to GDI and USER, shutting down Windows for the duration between the Lock and Unlock operations, as well as between the GetDC and ReleaseDC operations.
- Be sure you 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. This module 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 happens to span memory banks.

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 methods will return DDERR_SURFACELOST and perform no other function.

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 IDirectDrawSurface::Restore method recreates these lost surfaces and reconnects them to their DirectDrawSurface objects.

Surface Format Support in the HEL

The following table shows the pixel formats for off-screen plain surfaces supported by the DirectX 2 HEL. The "Masks" column shows the red, green, blue, and alpha masks for each set of pixel format flags and bit depth.

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.

Color and Format Conversion

Non-RGB surface formats are described by FourCC codes. When the pixel format is requested, if the surface is a non-RGB surface, the DDPF_FOURCC flag will be set and the **dwFourCC** member in 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. For both of these types of color keying, a color key or a color range may be supplied.

Source color keying specifies a color or color range that, in the case of blitting, will not be copied, or, in the case of overlays, not be visible on the destination. Destination color keying specifies a color or color range that, in the case of blitting, will be replaced or, in the case of overlays, be covered up on the destination. The source color key specifies what can and can't be read from the source surface. The destination color key specifies what can and can't 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 will be changed, or covered up, on the destination surface.

Some hardware will only support color ranges for YUV pixel data. This is because YUV data is usually video, and, due to quantitization errors during conversion, the transparent background is not a single color.

Content should be authored to a single transparent color whenever possible, regardless of pixel format.

Specifying Color Keys

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 has the V data, the second lowest order byte has the U data, and the highest order byte has the Y data. The IDirectDrawSurface::SetColorKey method has a flags parameter that specifies whether the color key is to be used for overlay operations 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);
```

Flipping Surfaces and GDI's Frame Rate

DirectDraw has extended flipping surfaces to encompass more than page flipping and more than visible surface flipping. Any surface can now be constructed as a flipping surface. This has many advantages over the traditional, limited scope of page flipping.

When an IDirectDrawSurface::Flip method operation is requested in DirectDraw, 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 just switched with the front buffer when the **IDirectDrawSurface::Flip** method 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 IDirectDrawSurface::Lock method or the IDirectDrawSurface::Blt method that target the back buffer will fail with the DDERR_WASSTILLDRAWING return value until the next vertical refresh occurs. This behavior 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 **IDirectDrawSurface::Flip** method on visible surfaces an asynchronous command. A good practice to follow when building games, for example, is to 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, 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, and when a surface that GDI is not aware of is about to become the primary surface, 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 a 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. Since GDI has no knowledge of DirectDraw, GDI will continue operating on this surface, even if you have flipped it and it is now the non-visible back buffer.

Many applications will 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 since nothing it controls needs redrawing.

For other scenarios, always remember that GDI only knows about the original surface, and never knows if it is currently the primary surface or a back buffer. If you do not need the GDI screen, then use the above technique. If you do need GDI, you can try this technique:

- Create a primary surface with two back buffers.
- Blit the initial primary surface (the GDI surface) to the middle back buffer.
- Flip(NULL) to put GDI into last place and make your initial copy visible.

After you have done this, copy from the GDI buffer to the middle buffer, draw what you want the user to see on that buffer, then use the code below, which keeps GDI safely on the bottom and oscillates between the other two buffers.

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

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 only affected by the bits on the primary surface, not by overlays occluded by other overlays. Source color keys work on an overlay whether or not it has a z-order specified. Overlays that do not have a specified z-order behave in unspecified 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 begins with 0, just on top of the primary surface, and ends with 4 billion, just underneath the glass on the monitor. An overlay with a z-order of 2 would obscure an overlay with a z-order of 1. An overlay is not allowed to have the same z-order as another overlay.

Palettes and Pixel Formats

DirectDraw enables the creation of 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 an off-screen surface with a pixel format different from the primary surface is created, it is assumed that the hardware can use it. For instance, if a paletized 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 supports the creation of standard 8-bit palettized surfaces, capable of displaying 256 colors, and two kinds of 4-bit palettized surfaces, each capable of displaying 16 colors. The first kind of 4-bit palettized surface is indexed into a true color palette table; the second kind is indexed into the primary surface indexed palette table. 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, while blitting from one palettized surface to another, the palette is ignored. Palette decoding is only done 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.

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. The DDCAPS structure has been expanded to allow drivers to report this capability. The following members have been added:

```
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   dwSSBFXCaps
DWORD   dwSSBRops[DD_ROP_SPACE]
```

The SVB prefix indicates capabilities values that relate system memory to display memory blits. The VSB prefix indicates capabilities values that relate display memory to system memory blits. The SSB prefix indicates capabilities values that relate 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. Likewise, 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 only valid 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 will automatically call the IDirectDrawSurface2::PageLock method on the surface to ensure that the memory has been locked.

Setting Palettes on Non-Primary Surfaces

In DirectX 2, 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 DirectDraw surface method, IDirectDrawSurface::GetDC, also ignores any DirectDrawPalette selected into the surface.

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

Sharing Palettes

In DirectX 2, palettes can be shared between multiple surfaces. The same palette can be set on the front and back buffers of a flipping chain or shared between multiple texture surfaces. When a palette is attached to a surface with the IDirectDrawSurface::SetPalette method, the surface increments the reference count of that palette. When the reference count of the surface reaches 0, it will decrement the reference count of the attached palette. In addition, if a palette is detached from a surface by calling **IDirectDrawSurface::SetPalette** with a NULL palette interface pointer, the reference count of the surface's palette will be decremented.

Note If the **IDirectDrawSurface::SetPalette** method is called several times consecutively for the same surface with the same palette, the reference count for the palette will be incremented only once. Subsequent calls will not affect the palette's reference count.

New Palette Types

In DirectX 2, DirectDraw supports 1-bit (2 entry), 2-bit (4 entry) and 4-bit (16 entry) palettes in addition to the 8-bit (256 entry) palettes supported by previous versions. Such palettes can be created by specifying one of the new palette capability flags: DDPCAPS_1BIT, DDPCAPS_2BIT, and DDPCAPS_4BIT. Matching capability flags have been added for surface pixel formats: DDPF_PALETTEINDEXED1, DDPF_PALETTEINDEXED2, and DDPF_PALETTEINDEXED4.

A palette can only be attached to a surface with a matching pixel format. For example, a 2 entry palette created with the DDPCAPS_1BIT flag can only be attached to a 1-bit surface created with the pixel format flag DDPF_PALETTEINDEXED1.

Furthermore, it is now 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 PALETTEENTRYs 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 palette capability flag DDPCAPS_8BITENTRIES when calling the IDirectDraw::CreatePalette method. For example, to create a 4-bit, indexed palette, specify DDPCAPS_4BIT | DDPCAPS_8BITENTRIES. When creating an indexed palette, a pointer to an array of bytes is passed rather than a pointer to an array of PALETTEENTRY structures. The pointer to the array of bytes must be cast to an LPPALETTEENTRY when calling **IDirectDraw::CreatePalette**.

Driver Independent Clippers

You can create clipper objects that are not directly owned by any particular DirectDraw object. Such clipper objects can be shared across multiple DirectDraw objects. Driver independent clipper objects are created with the new DirectDraw API function DirectDrawCreateClipper. This function can be called before any DirectDraw objects are created.

Because these clippers are not owned by any DirectDraw object, they are not automatically released when your application's objects are released. If not released explicitly by the application, these clippers will be released by DirectDraw when the application terminates.

You can still create clippers with the IDirectDraw::CreateClipper method. These DirectDrawClipper objects are automatically released when the DirectDraw object from which they were created is released.

Clip Lists

Clip lists are managed by DirectDraw using the DirectDrawClipper object. A DirectDrawClipper can be attached to any surface. A window handle can also be attached to a DirectDrawClipper, in which case DirectDraw will update 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 will only call the HAL 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 blit would be the upper left hand corner of the surface and the second would be 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 being destination color keyed. 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 Clippers

In DirectX 2, clippers can be shared between multiple surfaces. For example, the same clipper can be set on both the front and back buffers of a flipping chain. When a clipper is attached to a surface by using the IDirectDrawSurface::SetClipper method, the surface increments the reference count of that clipper. When the reference count of the surface reaches 0, it will decrement the reference count of the attached clipper. In addition, if a clipper is detached from a surface by calling **IDirectDrawSurface::SetClipper** with a NULL clipper interface pointer, the reference count of the surface's clipper will be decremented.

Note If **IDirectDrawSurface::SetClipper** is called several times consecutively on the same surface for the same clipper, the reference count for the clipper will be incremented only once. Subsequent calls will not affect the clipper's reference count.

Creating Clipper Objects Using CoCreateInstance

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

CLSID_DirectDrawClipper is the class identifier of the DirectDrawClipper object class, **IID_IDirectDrawClipper** is the currently supported interface (which is the one you want), and *lpClipper* is the clipper object returned.

Clippers created by the class factory mechanism must be initialized with the **IDirectDrawClipper::Initialize** method before you can use the object. 0UL is the *dwFlags* parameter, which in this case has a value of 0 since 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 instead, which would create an independent clipper (equivalent to creating a DirectDrawClipper object using the **DirectDrawCreateClipper** function).

Before closing the application, shut down COM using **CoUninitialize**, as shown below.

```
    CoUnitialize();
```

Texture Maps

In DirectX 2, texture maps can be allocated in system memory using the HEL. To allocate a texture map surface, specify the DDSCAPS_TEXTURE flag in the **ddsCaps** member of the surface description passed to the IDirectDraw::CreateSurface method.

A wide range of texture pixel formats are supported by the HEL. The following table describes 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 | DDPF_PALETTEINDEXED1	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED1 | DDPF_PALETTEINDEXEDTO8	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_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 formats shown in the previous table are those that can be created by the HEL in system memory. The DirectDraw device driver for a 3D-accelerated display card is free to create textures of other formats in display memory. Such a driver should export the DDSCAPS_TEXTURE flag to indicate that it can create textures, and should be prepared to handle the DirectDraw HAL callback **CanCreateSurface** to verify that the surface description for a texture map is one the driver is prepared to create.

Mipmaps

In DirectX 2, DirectDraw supports mipmapped texture surfaces. A mipmap is a sequence of textures, each of which is a progressively lower resolution, prefiltered representation of the same image. Mipmapping is a computationally low-cost way of improving the quality of rendered textures. Each pre-filtered image, or level, in the mipmap is a power of two smaller than the previous level. 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 which 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 surface description passed to the IDirectDraw::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 with the IDirectDrawSurface::AddAttachedSurface method. However, the **IDirectDraw::CreateSurface** method can be used 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;
LPDIRECTDRAWSURFACE 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 mipmaps levels, in which case the IDirectDraw::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 **IDirectDraw::CreateSurface** will create the number of levels you specify with a minimum level size of 1×1.

A chain of mipmap surfaces is traversed using the IDirectDrawSurface::GetAttachedSurface method, specifying the DDSCAPS_MIPMAP and DDSCAPS_TEXTURE capability flags. The following example traverses a mipmap chain from highest to lowest resolutions.

```
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 flippable chains of mipmaps. In this scenario, each mipmap level has an associated chain of back buffer texture surfaces. Each back buffer texture 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 (DDSCAPS_TEXTURE). 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 flippable chain. All the surfaces in each flippable chain must be of the same size.

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

Note 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 IDirectDrawSurface::Flip method will flip all the levels of a mipmap from the level supplied to the lowest level in the map. A destination surface can also be provided, in which case all levels in the mipmap will flip to the back buffer in their flippable chain. This back buffer matches the supplied override. For example, if the third back buffer in the top-level flippable chain is supplied, all levels in the mipmap will flip to their third back buffer.

The number of levels in a mipmap chain is stored explicitly. When the surface description of a mipmap is obtained (using IDirectDrawSurface::Lock or IDirectDrawSurface::GetSurfaceDesc), the dwMipMapCount member will contain the number of levels in a mipmap, including the top level. For levels other than the top level in the map, **dwMipMapCount** will specify the number of levels from that map to the smallest map in the chain.

Z-Buffers

In DirectX 2, 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 surface capability DDSCAPS_ZBUFFER. It should also specify the z-buffer depths it supports using the **dwZBufferBitDepths** member of the DDCAPS structure.

Z-buffers can be cleared using the IDirectDrawSurface::Blt method. A new DirectDraw blit flag (DDBLT_DEPTHFILL) has been defined to indicate that the blit clears z-buffers. If this flag is specified, the DDBLTFX structure passed to the **IDirectDrawSurface::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 capability flag DDCAPS_BLTDEPTHFILL and should have code to handle DDBLT_DEPTHFILL blits. The destination surface of a depth fill blit must be a z-buffer.

Note The actual interpretation of a depth value is 3D-renderer specific.

Direct3D Driver Interface

DirectDraw presents a single, unified object to you as an application programmer. This object encapsulates both the DirectDraw and Direct3D states. The DirectDraw driver COM interfaces (IID_IDirectDraw or IID_IDirectDraw2) and the Direct3D driver COM interface (IID_IDirect3D) all allow you to communicate with the same underlying object. Therefore, no Direct3D object is created. Rather, a Direct3D interface to the DirectDraw object is obtained. This is achieved using the standard COM **QueryInterface** method.

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 after the IDirectDraw::QueryInterface method call is two. 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, whether DirectDraw or Direct3D, to that object have been released. Hence, if you release a Direct3D interface while holding a reference to a DirectDraw driver interface, and re-query the Direct3D interface, the Direct3D state will be preserved.

Direct3D Device Interface

As with the object, there is no distinct Direct3D device object. A Direct3D device is simply an interface for communicating with a DirectDraw surface 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 (see Direct3D Driver Interface) apply to DirectDraw surfaces and Direct3D devices. Additionally, multiple, distinct Direct3D device interfaces can be obtained for the same DirectDraw surface. It is possible, therefore, that a single DirectDraw surface 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 discussed for objects (see Direct3D Driver Interface) apply to Direct3D textures. It is possible to use a single DirectDrawSurface as both a rendering target and a texture.

DirectDraw HEL and Direct3D

The DirectDraw HEL has been enhanced to support the creation of texture, mipmap, and z-buffer surfaces. Furthermore, due to the tight integration of DirectDraw and Direct3D, a DirectDraw-enabled system will always provide Direct3D support (in software emulation, at least). Hence, the DirectDraw HEL exports the DDSCAPS_3DDEVICE flag to indicate that a surface can be used for 3D rendering. A DirectDraw driver for a hardware-accelerated 3D display card should export this capability to indicate the presence of hardware-accelerated 3D.

Functions

Two of the DirectDraw functions initiate control of display memory through the IDirectDraw interface. The first function, DirectDrawCreate, creates an instance of a DirectDraw object that represents a specific piece of display hardware. The second function, DirectDrawEnumerate, obtains a list of all DirectDraw objects installed on a system. These functions are the mechanisms DirectDraw uses to support multiple pieces of display hardware. To support multiple display devices, your application need only select a specific DirectDraw object and instantiate it.

The third function, DirectDrawCreateClipper, creates an instance of a DirectDraw clipper that is not directly owned by a DirectDraw object, and can be shared across multiple driver objects.

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 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** will return an error if this parameter is anything but NULL.

This function attempts to initialize a DirectDraw object, then sets a pointer to the object if successful. Calling the IDirectDraw::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.

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

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

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

lplpDDClipper
Address of a pointer to be filled in with the address of the new DirectDrawClipper object.

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

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

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 a caller-defined context that will be passed to the enumeration callback each time it is called.

Callback Functions

Most of the functionality of DirectDraw is provided by the methods of its Component Object Model (COM) interfaces. This section lists the callback functions that are not implemented as part of a COM interface.

Callback

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

Application-defined callback procedure 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 a caller-defined structure that will be passed to the callback function each time the function is invoked.

EnumCallback

```
HRESULT WINAPI lpEnumCallback(LPDIRECTDRAWSURFACE lpDDSurface,
    LPDDSURFACEDESC lpDDSurfaceDesc,LPVOID lpContext);
```

Application-defined callback procedure for the IDirectDraw::EnumSurfaces method.

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

lpDDSurface

Address of the DirectDrawSurface currently being enumerated if it is an *existing* surface (DDENUMSURFACES_DOESEXIST). The value will be NULL if a *potential* surface is being enumerated (DDENUMSURFACES_CANBECREATED).

lpDDSurfaceDesc

Address of the DDSURFACEDESC structure for the existing or potential surface that most closely matches the requested surface.

lpContext

Address of the caller-defined structure passed to the member every time it is invoked.

EnumModesCallback

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

Application-defined callback procedure 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 a caller-defined structure that will be passed to the callback function each time the function is invoked.

EnumSurfacesCallback

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

Application-defined callback procedure for the IDirectDrawSurface::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 the user-defined context specified by the user.

fnCallback

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

Application-defined callback procedure for the IDirectDrawSurface::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 the user-defined context specified by the user.

IDirectDraw Interface

DirectDraw objects represent the display hardware. An object is hardware-accelerated if the display device for which it was instantiated has hardware acceleration. Three types of objects can be created by a DirectDraw object: DirectDrawSurface, DirectDrawPalette, and DirectDrawClipper.

More than one DirectDraw object can be instantiated at a time. The simplest example of this would be using two monitors on a Windows 95 system. Although Windows 95 does not support dual monitors natively, 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 does not recognize can be addressed by another, independent DirectDraw object that must be created using the second display device's identifying GUID. This GUID can be obtained through 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 bpp mode, the default color key values, and the hardware display mode. It tracks what resources have been allocated and what resources remain to be allocated.

Changing the display mode is an important piece of DirectDraw functionality. The display mode resolution can be changed at any time unless another application has obtained exclusive access to DirectDraw. The pixel depth of the display mode can only be changed if the application requesting the change 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 using the IDirectDrawSurface::Restore method.

IDirectDraw Interface Method Groups

Applications use the methods of the IDirectDraw interface to create DirectDraw objects and work with system-level variables. The methods 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
IUnknown	AddRef QueryInterface Release
Miscellaneous	GetAvailableVidMem GetFourCCCodes WaitForVerticalBlank
Setting behavior	SetCooperativeLevel
Surfaces	DuplicateSurface EnumSurfaces FlipToGDISurface GetGDISurface

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectDraw object without affecting the functionality of the original interface.

IDirectDraw::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectDraw object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

When the DirectDraw object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirectDraw::Release method to decrease the reference count of the object by 1.

IDirectDraw::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_SURFACEBUSY DDERR_NOEXCLUSIVEMODE

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 will fail if any operations are in progress.

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

IDirectDraw::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_OUTOFMEMORY DDERR_NOCOOPERATIVELEVELSET

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

lplpDDClipper
Address of a pointer to be filled in with the address of the new DirectDrawClipper object if the **IDirectDraw::CreateClipper** method is successful.

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

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

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

IDirectDraw::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_NOEXCLUSIVEMODE
DDERR_INVALIDPARAMS	DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFCAPS	DDERR_OUTOFMEMORY
DDERR_UNSUPPORTED	

dwFlags

DDPCAPS_1BIT

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

DDPCAPS_2BIT

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

DDPCAPS_4BIT

Indicates the index is 4 bits. There are sixteen entries in the palette table.

DDPCAPS_8BITENTRIES

An index to an 8-bit color index. This flag is only valid 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 one byte long and is an index to a destination surface's 8-bpp palette.

DDPCAPS_8BIT

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

DDPCAPS_ALLOW256

Indicates 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 to be filled in with the address of the new DirectDrawPalette object if the **IDirectDraw::CreatePalette** method is successful.

pUnkOuter

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

IDirectDraw::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

lpSurfaceDesc
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, **IDirectDraw::CreateSurface** will return an error if this parameter is anything but NULL.

The DirectDrawSurface object represents a surface (pixel memory) that usually resides in the display card memory, but 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, the call to **IDirectDraw::CreateSurface** will fail.

This method usually creates one DirectDrawSurface object. If the DDSCAPS_FLIP flag in the **dwCaps** member of the DDSCAPS structure is set, **IDirectDraw::CreateSurface** will create several DirectDrawSurface objects, referred to collectively as a *complex structure*. The additional surfaces created are also referred to as *implicit* surfaces.

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 *dwHeight* and *dwWidth* values, 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 in.

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

//Tell DDRAW which fields are valid
ddsd.dwFlags = DDSD_CAPS;

//Ask for 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 in the **DDSURFACEDESC** structure (*ddsd* below) relevant to the creation of a simple off-screen surface are then filled in.

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

//Tell DDRAW which fields are valid
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;

//Ask for 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 flags DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY in **dwCaps** 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 **IDirectDraw::CreateSurface** method. If the DDSCAPS_COMPLEX flag is set in the **IDirectDraw::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 will release all surfaces in the structure, and a single call to the IDirectDrawSurface::Restore method will restore them all.

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 in 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 DDRAW which fields are valid
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;

//Ask for 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 IDirectDrawSurface::Flip method exchanges the surface memory of the primary surface and the back buffer. If a *BackBufferCount* of 2 is specified, two back buffers are created, and each call to

IDirectDrawSurface::Flip rotates the surfaces in a circular pattern, providing a triple-buffered flipping environment.

IDirectDraw::DuplicateSurface

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

Duplicates a DirectDrawSurface.

- 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 will be enumerated.

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

dwFlags

DDEDM_REFRESHRATES

Enumerate 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.

DDSD_REFRESHRATE

The **dwRefreshRate** member of the DDSURFACEDESC structure is valid.

lpDDSurfaceDesc

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

lpContext

Address of a caller-defined structure that will be passed to each enumeration member.

lpEnumModesCallback

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

This method enumerates the **dwRefreshRate** member in the **DDSURFACEDESC** structure; the **IDirectDraw2::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.

To ensure COM compliance, this method is not part of the IDirectDraw interface, but belongs to the IDirectDraw2 interface. To use this method, you must first query for the IDirectDraw2 interface. For more information, see IDirectDraw2 Interface.

IDirectDraw::EnumSurfaces

```
HRESULT EnumSurfaces(DWORD dwFlags,
    LPDDSURFACEDESC lpDDSD, LPVOID lpContext,
    LPDDENUMSURFACESCALLBACk lpEnumCallback);
```

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

DDENUMSURFACES_ALL

Enumerates all of the 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.

DDENUMSURFACES_CANBECREATED

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

DDENUMSURFACES_DOESEXIST

Enumerates the already existing surfaces that meet the search criterion.

lplpDDSD

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

lpContext

Address of a caller-defined structure to be passed to each enumeration member.

lpEnumCallback

Address of the EnumCallback function the enumeration procedure will call every time a match is found.

If the DDENUMSURFACES_CANBECREATED flag is set, this method will attempt to temporarily create a surface that meets the criteria. 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.

IDirectDraw::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.

IDirectDraw2::GetAvailableVidMem

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

Gets the total amount of display memory available and the amount of display memory currently free. If NULL is passed to either *lpdwTotal* or *lpdwFree*, the value for that parameter is not returned.

- Returns DD_OK if successful, or one of the following error values otherwise:
  DDERR_INVALIDOBJECT DDERR_NODIRECTDRAWHW
  DDERR_INVALIDPARAMS DDERR_INVALIDCAPS

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

lpdwTotal
Address of a doubleword to be filled in with the total amount of display memory available.

lpdwFree
Address of a doubleword to be filled in with the amount of display memory currently free.

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 only gives 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, the free memory value should only be used as a rough guide. In addition, a particular display adapter card may make no distinction between two different memory types. For example, it may use the same portion of display memory to store z-buffers and textures. Hence, allocating one type of surface (for example, a z-buffer) may 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, back and z-buffers) before determining how much memory is available for dynamic use (such as texture mapping).

To ensure COM compliance, this method is not a member of the IDirectDraw interface, but is part of the IDirectDraw2 interface. To use this method, you must first query for the IDirectDraw2 interface. For more information, see IDirectDraw2 Interface.

IDirectDraw::GetCaps

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

Fills in the raw, not remaining, 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 to be filled in with the capabilities of the hardware, as reported by the device driver.

lpDDHELCaps
Address of a **DDCAPS** structure to be filled in with the capabilities of the HEL.

IDirectDraw::GetDisplayMode

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

Returns 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 to be filled in with a description of the surface.

An application should not save the information returned by **IDirectDraw::GetDisplayMode** to restore the display mode on clean up. The mode restoration on clean up should be performed with IDirectDraw::RestoreDisplayMode, thereby avoiding mode setting conflicts that could arise in a multiprocess environment.

IDirectDraw::GetFourCCCodes

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

Returns the FourCCCodes supported by the DirectDraw object. **IDirectDraw::GetFourCCCodes** can also return 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 doubleword that contains the number of entries the *lpCodes* array can hold. If the number of entries is too small to accommodate all the codes, *lpNumCodes* will be set to the required number and the *lpCodes* array will be filled with all that will fit.

lpCodes

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

IDirectDraw::GetGDISurface

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

Returns the DirectDrawSurface object that currently represents the surface memory 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.

IDirectDraw::GetMonitorFrequency

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

Returns 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 doubleword to be filled in with the monitor frequency.

IDirectDraw::GetScanLine

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

Returns the scan line 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 doubleword to contain the scan line the display is currently on.

IDirectDraw::GetVerticalBlankStatus

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

Returns 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 BOOL to be filled in with the status of the vertical blank.

This method will set the passed BOOL to TRUE if a vertical blank is occurring and to FALSE otherwise. To synchronize with the vertical blank, use the IDirectDraw::WaitForVerticalBlank method.

IDirectDraw::Initialize

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

Initializes the DirectDraw object.

- Returns DDERR_ALREADYINITIALIZED.

lpGUID

Address of the GUID used as the interface identifier.

This method is provided for compliance with the Component Object Model (COM) protocol. Since the DirectDraw object is initialized when it is created, calling this method will always result in the DDERR_ALREADYINITIALIZED return value.

IDirectDraw::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID FAR * ppvObj);
```

Determines if the DirectDraw object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by DirectDraw.

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

riid
Reference identifier of the interface being requested.

ppvObj
Address of a pointer to be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectDraw::QueryInterface** method allows DirectDraw objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirectDraw::Release

```
ULONG Release();
```

Decreases the reference count of the DirectDraw object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

The DirectDraw object deallocates itself when its reference count reaches 0. Use the IDirectDraw::AddRef method to increase the reference count of the object by 1.

IDirectDraw::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 to change it. 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

IDirectDraw::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_HWNDALREADYSET	DDERR_INVALIDPARAMS
DDERR_HWNDSUBCLASSED	DDERR_INVALIDOBJECT
DDERR_EXCLUSIVEMODEALREADYSET	DDERR_OUTOFMEMORY

hWnd

Specifies the window handle used for the application.

dwFlags

DDSCL_ALLOWMODEX

Allows the use of ModeX display modes.

DDSCL_ALLOWREBOOT

Allows CTRL_ALT_DEL to function while in full screen exclusive 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 IDirectDraw::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 **IDirectDraw::SetCooperativeLevel** method with DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags set, DirectDraw will attempt to resize its window to full screen. An application must either set the DDSCL_EXCLUSIVE or DDSCL_NORMAL flags, and DDSCL_EXCLUSIVE requires DDSCL_FULLSCREEN.

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

ModeX modes are not supported by Windows; therefore, when in a ModeX mode you cannot use the IDirectDrawSurface::Lock method to lock the primary surface, or the IDirectDrawSurface::Blt method to blit to the primary surface. Use the IDirectDrawSurface::GetDC method on the primary surface, or use GDI with a screen DC. ModeX modes are indicated by the DDSCAPS_MODEX flag in the DDSCAPS member of the DDSURFACEDESC structure returned by the IDirectDrawSurface::GetCaps and **IDirectDraw2::EnumDisplayModes** methods.

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

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
Specifies the width of the new mode.

dwHeight
Specifies the height of the new mode.

dwBPP
Specifies the bits per pixel of the new mode.

dwRefreshRate
Specifies the 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 not currently used and must be set to 0.

The IDirectDraw::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 will return DDERR_SURFACELOST until they are restored.

To ensure COM compliance, this method is not part of the IDirectDraw interface, but belongs to the IDirectDraw2 interface. To use this method, you must first query for the IDirectDraw2 interface. For more information, see IDirectDraw2 Interface.

IDirectDraw::WaitForVerticalBlank

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

Helps the caller 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 is not currently supported.

DDWAITVB_BLOCKEND

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

hEvent

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

IDirectDrawSurface Interface

The DirectDrawSurface object represents a two-dimensional 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 the IDirectDraw::CreateSurface method. Although it is not required, the DirectDrawSurface object usually resides in the display RAM of the display card. Unless specifically stated during DirectDrawSurface object creation, 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 central processing unit.

DirectDrawSurface objects recognize, and are integrated with, the rest of the components of the Windows display system. DirectDrawSurface objects can create handles to Window 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.

IDirectDrawSurface Interface Method Groups

Applications use the methods of the IDirectDrawSurface interface to create DirectDrawSurface objects and work with system-level variables. The methods 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
IUnknown	AddRef QueryInterface Release
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

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectDrawSurface object without affecting the functionality of the original interface.

IDirectDrawSurface::AddAttachedSurface

```
HRESULT AddAttachedSurface(
    LPDIRECTDRAWSURFACE 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 DirectDrawSurface that is to be attached.

Possible attachments include z-buffers, alpha channels, and back buffers. Some attachments automatically break other attachments. For example, the 3DZBUFFER can only be attached to one back buffer at a time. Attachment is not bi-directional, 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 flippable surface cannot be attached to another flippable surface of the same type; however, attaching two surfaces of different types is allowed. For example, a flippable z-buffer can be attached to a regular flippable surface. If a non-flippable surface is attached to another non-flippable surface of the same type, the two surfaces will become a flippable chain. If a non-flippable surface is attached to a flippable surface, it becomes part of the existing flippable chain. Additional surfaces can be added to this chain, and each call of the IDirectDrawSurface::Flip method will advance one step through the surfaces.

IDirectDrawSurface::AddOverlayDirtyRect

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

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

- Returns DD_OK if successful, or one of the following error values otherwise:
  DDERR_INVALIDOBJECT DDERR_INVALIDPARAMS
  DDERR_UNSUPPORTED DDERR_INVALIDSURFACETYPE

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.

IDirectDrawSurface::AddRef

```
ULONG  AddRef();
```

Increases the reference count of the DirectDrawSurface object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

When the DirectDraw object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirectDrawSurface::Release method to decrease the reference count of the object by 1.

IDirectDrawSurface::Blt

```
HRESULT Blt(LPRECT lpDestRect,
    LPDIRECTDRAWSURFACE 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 DirectDrawSurface structure that represents the DirectDrawSurface. This 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 in the DDBLTFX structure as the alpha channel for the destination surface for this blit.

DDBLT_ALPHADESTNEG

The NEG suffix indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHADESTSURFACEOVERRIDE

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

DDBLT_ALPHAEDGEBLEND

Uses the **dwAlphaEdgeBlend** member in 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 in the **DDBLTFX** structure as the alpha channel for the source for this blit.

DDBLT_ALPHASRCNEG

The NEG suffix indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHASRCSURFACEOVERRIDE

Uses the **lpDDSAlphaSrc** member in 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, fail the call.

DDBLT_COLORFILL

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

DDBLT_DDFX

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

DDBLT_DDROPS

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

DDBLT_DEPTHFILL

Uses the **dwFillDepth** member in 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 in 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 in the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP

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

DDBLT_ROTATIONANGLE

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

DDBLT_WAIT

Postpones the DDERR_WASSTILLDRAWING return value if the blitter is busy. 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 in the **DDBLTFX** structure as the z-buffer opcode.

DDBLT_ZBUFFERDESTCONSTOVERRIDE

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

DDBLT_ZBUFFERDESTOVERRIDE

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

lpDDBltFx

See 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 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, **IDirectDrawSurface::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.

IDirectDrawSurface::BltBatch

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

Performs a sequence of IDirectDrawSurface::Blt operations from several sources to a single destination. 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_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
The number of blit operations to be performed.

dwFlags
This parameter is not used at this time and must be set to 0.

IDirectDrawSurface::BltFast

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

Performs a source copy blit or transparent blit using a source or destination color key. This method always attempts an asynchronous blit if this 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

X-coordinate to blit to on the destination surface.

dwY

Y-coordinate to blit to on the destination surface.

lpDDSrcSurface

Address of the DirectDrawSurface structure that represents the DirectDrawSurface. This 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

Specifies the type of transfer.

DDBLTFAST_DESTCOLORKEY

A transparent blit that uses the destination's color key.

DDBLTFAST_NOCOLORKEY

A normal copy blit with no transparency.

DDBLTFAST_SRCCOLORKEY

A transparent blit that uses the source's color key.

DDBLTFAST_WAIT

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

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

Typically, **IDirectDrawSurface::BltFast** returns immediately with an error if the blitter is busy and the blit cannot be set up. The DDBLTFAST_WAIT flag can be used to alter this behavior so that this method will not return until either the blit can be set up or another error occurs.

IDirectDrawSurface::DeleteAttachedSurface

```
HRESULT DeleteAttachedSurface(DWORD dwFlags,
    LPDIRECTDRAWSURFACE 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_INVALIDOBJECT	DDERR_CANNOTDETACHSURFACE
DDERR_INVALIDPARAMS	DDERR_SURFACENOTATTACHED
DDERR_SURFACELOST	

dwFlags

This parameter is not used at this time and must be set to 0.

lpDDSAttachedSurface

Address of the DirectDrawSurface structure to be detached. If NULL is passed, all attached surfaces will be detached.

If NULL is passed as the surface to be detached, all attached surfaces will be detached. Implicit attachments, those formed by DirectDraw rather than the IDirectDrawSurface::AddAttachedSurface method, cannot be detached. Detaching surfaces from a flippable chain can alter other surfaces in the chain. If a front buffer is detached from a flippable 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 flippable chain only has two surfaces and they are detached, the chain is destroyed and both surfaces return to their previous designations.

IDirectDrawSurface::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 caller-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.

IDirectDrawSurface::EnumOverlayZOrders

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

Enumerates the overlays 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

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.

IDirectDrawSurface::Flip

```
HRESULT Flip(
    LPDIRECTDRAWSURFACE 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_NOTFLIPPABLE
  DDERR_NOFLIPHW
  DDERR_SURFACEBUSY
  DDERR_SURFACELOST
  DDERR_WASSTILLDRAWING
  DDERR_UNSUPPORTED

lpDDSurfaceTargetOverride

Address of the DirectDrawSurface structure that will be flipped to. The default for this parameter is NULL, in which case **IDirectDrawSurface::Flip** cycles through the buffers in the order they are attached to each other. This parameter is only used 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 error DDERR_WASSTILLDRAWING will return immediately and no flip will occur. Setting this flag causes **IDirectDrawSurface::Flip** to continue trying to flip if it receives the DDERR_WASSTILLDRAWING error from the HAL. **IDirectDrawSurface::Flip** will not return until the flipping operation has been successfully set up, or if another error, such as DDERR_SURFACEBUSY, is returned.

This method can only be called 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 **IDirectDrawSurface::Flip** is invoked.

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

IDirectDrawSurface::GetAttachedSurface

```
HRESULT GetAttachedSurface(LPDDSCAPS lpDDSCaps,
    LPLPDIRECTDRAWSURFACE 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 DirectDrawSurface that will be attached to the current DirectDrawSurface specified by *lpDDSurface* and 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 will fail if more than one surface is attached that matches the capabilities requested. In this case, the application must use the IDirectDrawSurface::EnumAttachedSurfaces method to obtain the non-unique attached surfaces.

IDirectDrawSurface::GetBltStatus

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

Obtains the blitter status. This method returns DD_OK if a blitter is present, DDERR_WASSTILLDRAWING if the blitter is busy, or DDERR_NOBLTHW if there is no blitter.

- Returns DD_OK if successful, or one of the following error values otherwise:
  DDERR_INVALIDOBJECT DDERR_INVALIDPARAMS
  DDERR_NOBLTHW DDERR_SURFACEBUSY
  DDERR_SURFACELOST DDERR_UNSUPPORTED
  DDERR_WASSTILLDRAWING

dwFlags

DDGBS_CANBLT

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

DDGBS_ISBLTDONE

Inquires whether the blit is done. Returns DD_OK if the last blit on this surface has completed.

IDirectDrawSurface::GetCaps

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

Returns 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

lpDDCaps
Address of a DDCAPS structure that will be filled in with the hardware capabilities of the surface.

IDirectDrawSurface::GetClipper

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

Returns the DirectDrawClipper associated with this surface. An error returns if no DirectDrawClipper is associated.

- 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 associated with the surface.

IDirectDrawSurface::GetColorKey

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

Returns 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_NOCOLORKEYHW DDERR_NOCOLORKEY
  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 in with the current values for the specified color key of the DirectDrawSurface object.

IDirectDrawSurface::GetDC

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

Creates a GDI-compatible hDC 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 of the returned hDC.

This method uses an internal version of the IDirectDrawSurface::Lock method to lock the surface, and the surface will remain locked until the IDirectDrawSurface::ReleaseDC method is called. For more information, see the description of the **IDirectDrawSurface::Lock** method.

IDirectDrawSurface2::GetDDInterface

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

Returns 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.

To ensure COM compliance, this method is not part of the IDirectDrawSurface interface, but of the IDirectDrawSurface2 interface. To use this method, you must first query for the IDirectDrawSurface2 interface. For more information, see IDirectDrawSurface2 Interface.

IDirectDrawSurface::GetFlipStatus

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

Indicates whether the surface has finished its flipping process. If the surface has not finished its flipping process, it returns DDERR_WASSTILLDRAWING.

- Returns DD_OK if successful, or one of the following error values otherwise:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS
  DDERR_INVALIDSURFACETYPE
  DDERR_SURFACEBUSY
  DDERR_SURFACELOST
  DDERR_UNSUPPORTED
  DDERR_WASSTILLDRAWING

dwFlags

DDGFS_CANFLIP

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

DDGFS_ISFLIPDONE

Inquires whether the flip done. Returns DD_OK if the last flip on this surface has completed.

IDirectDrawSurface::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
Address of the x-display coordinate.

lplY
Address of the y-display coordinate.

IDirectDrawSurface::GetPalette

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

Returns 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. This pointer will be filled in with the address of the DirectDrawPalette structure associated with this surface. This will be set to NULL if no DirectDrawPalette is associated with this surface.

If no palette has been explicitly associated with this surface, it returns NULL for the associated palette. However, if this is the primary surface or a back buffer to the primary surface, it returns a pointer to the system palette if the primary surface is in 8-bpp mode.

IDirectDrawSurface::GetPixelFormat

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

Returns 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 in with a detailed description of the current pixel and color space format of the surface.

IDirectDrawSurface::GetSurfaceDesc

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

Returns 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 to be filled in with the current description of this surface.

IDirectDrawSurface::Initialize

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

Initializes a DirectDrawSurface.

- Returns DDERR_ALREADYINITIALIZED.

lpDD

Address of the DirectDraw structure that represents the DirectDraw object.

lpDDSurfaceDesc

Address of a DDSURFACEDESC structure to be filled in with the relevant details about the surface.

This method is provided for compliance with the Component Object Model (COM) protocol. Since the DirectDrawSurface object is initialized when it is created, calling this method will always result in the DDERR_ALREADYINITIALIZED return value.

IDirectDrawSurface::IsLost

```
HRESULT IsLost();
```

Determines if the surface memory associated with a DirectDrawSurface has been freed. If the memory has not been freed, this method will return DD_OK.

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

This method can be used to reallocate surface memory. When a DirectDrawSurface object loses its surface memory, most methods will return DDERR_SURFACELOST and perform no other function.

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

IDirectDrawSurface::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 to be filled with the relevant details about the surface.

dwFlags

DDLOCK_EVENT

Triggers the event when **IDirectDrawSurface::Lock** can return the surface memory pointer requested. Set if an event handle is being passed to **IDirectDrawSurface::Lock**. If multiple locks of this type are placed on a surface, events will be 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 is in progress, a DDERR_WASSTILLDRAWING error will be returned immediately. If this flag is set, however, **IDirectDrawSurface::Lock** will retry 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 to a system event that is triggered when the surface is ready to be locked.

Once the pointer is obtained, the surface memory can be accessed by your application until the corresponding the IDirectDrawSurface::Unlock method is called. Once this occurs, the pointer to the surface memory is no longer valid.

It is illegal to blit from a region of a surface that is locked. If a blit is attempted on a locked surface, the blit will return either a DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES error value.

Typically, **IDirectDrawSurface::Lock** will return immediately with an error when a lock cannot be obtained because a blit is in progress. The DDLOCK_WAIT flag can be set to continue trying to obtain a lock.

To prevent VRAM from being lost during access to a surface, DirectDraw holds the Win16 lock between **IDirectDrawSurface::Lock** and **IDirectDrawSurface::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 will stop Windows from running, so **IDirectDrawSurface::Lock**/**IDirectDrawSurface::Unlock** and IDirectDrawSurface::GetDC/IDirectDrawSurface::ReleaseDC periods should be kept short. Unfortunately, because Windows is stopped, GUI debuggers cannot be used in between **IDirectDrawSurface::Lock**/**IDirectDrawSurface::Unlock** or **IDirectDrawSurface::GetDC**/**IDirectDrawSurface::ReleaseDC** operations.

IDirectDrawSurface2::PageLock

```
HRESULT PageLock(DWORD dwFlags);
```

Prevents a system memory surface from being paged out while a blit using DMA transfers to or from system memory is in progress. 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.

Note The performance of the operating system could be negatively affected if too much memory is locked.

- 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 not used at this time and must be set to 0.

This method only works on system memory surfaces; it will not page lock a display memory surface or an emulated primary surface. If this method is called on a display memory surface, it will do nothing except return DD_OK.

To ensure COM compliance, this method is not a part of the IDirectDrawSurface interface, but belongs to the IDirectDrawSurface2 interface. To use this method, you must first query for the IDirectDrawSurface2 interface. For more information, see IDirectDrawSurface2 Interface.

IDirectDrawSurface2::PageUnlock

```
HRESULT PageUnlock(DWORD dwFlags);
```

Unlocks a system memory surface, allowing it to be paged out. 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.

- 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 not used at this time and must be set to 0.

This method only works on system memory surfaces; it will not page unlock a display memory surface or an emulated primary surface. If this method is called on a display memory surface, it will do nothing except return DD_OK.

To ensure COM compliance, this method is not a part of the IDirectDrawSurface interface, but belongs to the IDirectDrawSurface2 interface. To use this method, you must first query for the IDirectDrawSurface2 interface. For more information, see IDirectDrawSurface2 Interface.

IDirectDrawSurface::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID FAR * ppvObj);
```

Determines if the DirectDrawSurface object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by DirectDraw.

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

riid

Reference identifier of the interface being requested.

ppvObj

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectDrawSurface::QueryInterface** method allows DirectDrawSurface objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirectDrawSurface::Release

```
ULONG Release();
```

Decreases the reference count of the DirectDrawSurface object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

The DirectDrawSurface object deallocates itself when its reference count reaches 0. Use the IDirectDrawSurface::AddRef method to increase the reference count of the object by 1.

IDirectDrawSurface::ReleaseDC

```
HRESULT ReleaseDC(HDC hDC);
```

Releases the hDC previously obtained with the IDirectDrawSurface::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
The hDC previously obtained by **IDirectDrawSurface::GetDC**.

This method also unlocks the surface previously locked when the **IDirectDrawSurface::GetDC** method was called.

IDirectDrawSurface::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_INCOMPATIBLEPRIMARY
DDERR_IMPLICITLYCREATED	DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS	DDERR_NOEXCLUSIVEMODE
DDERR_OUTOFMEMORY	DDERR_UNSUPPORTED
DDERR_WRONGMODE	

Surfaces can be lost because the display card's 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. When a DirectDrawSurface object loses its surface memory, many methods will return DDERR_SURFACELOST and perform no other function. The **IDirectDrawSurface::Restore** method will reallocate surface memory and reattach it to the DirectDrawSurface object.

A single call to this method will restore a DirectDrawSurface's associated implicit surfaces (back buffers, and so on). An attempt to restore an implicitly created surface will result in an error. **IDirectDrawSurface::Restore** will not work across explicit attachments created using the IDirectDrawSurface::AddAttachedSurface method — each of these surfaces must be restored individually.

IDirectDrawSurface::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 that will be attached to the DirectDrawSurface. If NULL, the current clipper 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. Once a DirectDrawClipper has been attached and a clip list is associated with it, the clipper object will be used for the IDirectDrawSurface::Blt, IDirectDrawSurface::BltBatch, and IDirectDrawSurface::UpdateOverlay operations involving the parent DirectDrawSurface. This method can also detach a DirectDrawSurface's current clipper.

IDirectDrawSurface::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 has the new color key values for the DirectDrawSurface object.

IDirectDrawSurface::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
New x-display coordinate.

lY
New y-display coordinate.

IDirectDrawSurface::SetPalette

```
HRESULT SetPalette(LPDIRECTDRAWPALETTE lpDDPalette);
```

Attaches the specified DirectDrawPalette to a surface. The surface will use 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.

IDirectDrawSurface::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 IDirectDrawSurface::Lock method. Since it is possible to call **IDirectDrawSurface::Lock** multiple times for the same surface with different destination rectangles, this pointer ties the calls to the **IDirectDrawSurface::Lock** and **IDirectDrawSurface::Unlock** methods together.

IDirectDrawSurface::UpdateOverlay

```
HRESULT UpdateOverlay(LPRECT lpSrcRect,
    LPDIRECTDRAWSURFACE 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 DirectDrawSurface structure that represents the DirectDrawSurface. This is the 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 the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for the destination overlay.

DDOVER_ALPHADESTCONSTOVERRIDE

Uses the **dwAlphaDestConst** member in the DDOVERLAYFX structure as the destination alpha channel for this overlay.

DDOVER_ALPHADESTNEG

The NEG suffix indicates that the destination surface becomes more transparent as the alpha value increases. (0 is opaque).

DDOVER_ALPHADESTSURFACEOVERRIDE

Uses the **lpDDSAlphaDest** member in the **DDOVERLAYFX** structure as the alpha channel destination for this overlay.

DDOVER_ALPHAEDGEBLEND

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

DDOVER_ALPHASRC

Uses 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 in the **DDOVERLAYFX** structure as the source alpha channel for this overlay.

DDOVER_ALPHASRCNEG

The NEG suffix indicates that the source surface becomes more transparent as the alpha value increases.

DDOVER_ALPHASRCSURFACEOVERRIDE

Uses the **lpDDSAlphaSrc** member in the DDOVERLAYFX structure as the alpha channel source for this overlay.

DDOVER_DDFX

Uses the overlay FX flags to define special overlay FX.

DDOVER_HIDE

Turns this overlay off.

DDOVER_KEYDEST

Uses the color key associated with the destination surface.

DDOVER_KEYDESTOVERRIDE

Uses the **dckDestColorkey** member in 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 in the **DDOVERLAYFX** structure as the color key for the source surface.

DDOVER_SHOW

Turns this overlay on.

DDOVER_ZORDER

Uses the **dwZOrderFlags** member in the **DDOVERLAYFX** structure as the z-order for the display of this overlay. The **lpDDSRelative** member will be used if the **dwZOrderFlags** member is set to either DDOVERZ_INSERTINBACKOF or DDOVERZ_INSERTINFRONTOF.

lpDDOverlayFx

See the DDOVERLAYFX structure.

IDirectDrawSurface::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_UNSUPPORTED DDERR_INVALIDSURFACETYPE

dwFlags

Specifies the type of update to perform.

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.

IDirectDrawSurface::UpdateOverlayZOrder

```
HRESULT UpdateOverlayZOrder(DWORD dwFlags,
    LPDIRECTDRAWSURFACE lpDDSReference);
```

Sets the z-order of an overlay. The z-order determines which overlay should be occluded when multiple overlays are displayed simultaneously. Overlay positions are all relative to other overlays — there is no true z-value for them.

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

dwFlags

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 DirectDrawSurface structure that represents the DirectDrawSurface to be used as a relative position in the overlay chain. This parameter is needed only for DDOVERZ_INSERTINBACKOF and DDOVERZ_INSERTINFRONTOF.

IDirectDrawPalette Interface

The DirectDrawPalette object is provided to enable direct manipulation of 16- and 256-color palettes. DirectDrawPalette reserves entries 0 through 255 for 256-color palettes; however, it reserves no entries for 16-color palettes. It allows direct manipulation of the palette table as a table. 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.

Entries in these tables can be retrieved with the IDirectDrawPalette::GetEntries method and changed with the IDirectDrawPalette::SetEntries method. The **IDirectDrawPalette::SetEntries** method has a *dwFlags* parameter that specifies when the changes to the palette should take effect.

DirectDrawPalette objects are usually attached to DirectDrawSurface objects.

Two approaches can be used to provide straightforward palette animation using DirectDrawPalette objects. The first approach involves changing the palette entries that correspond to the colors that need to be animated. This can be done with a single call to the **IDirectDrawPalette::SetEntries** method. The second approach requires two DirectDrawPalette objects. The animation is performed by attaching first one object then the other to the DirectDrawSurface. This can be done using the IDirectDrawSurface::SetPalette method.

IDirectDrawPalette Interface Method Groups

Applications use the methods of the IDirectDrawPalette interface to create DirectDrawPalette objects and work with system-level variables. The methods can be organized into the following groups:

Allocating memory	Initialize
IUnknown	AddRef QueryInterface Release
Palette capabilities	GetCaps
Palette entries	GetEntries SetEntries

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectDrawPalette object without affecting the functionality of the original interface.

IDirectDrawPalette::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectDrawPalette object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

When the DirectDraw object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirectDrawPalette::Release method to decrease the reference count of the object by 1.

IDirectDrawPalette::GetCaps

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

Returns the capabilities of this palette object.

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

lpdwCaps
Flags from the **dwPalCaps** member of the DDCAPS structure that define 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.

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

dwFlags
This parameter is not used at this time and must be set to 0.

dwBase
Indicates the start of the entries that should be retrieved sequentially.

dwNumEntries
Indicates the number of palette entries *lpEntries* has room for. The colors of each palette entry will be returned in sequence, from *dwStartingEntry* through *dwCount* -1.

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

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
This parameter is not used at this time and must be set to 0.

lpDDColorTable
This parameter is not used at this time and must be set to 0.

This method is provided for compliance with the Component Object Model (COM) protocol. Since the DirectDrawPalette object is initialized when it is created, calling this method will always result in the DDERR_ALREADYINITIALIZED return value.

IDirectDrawPalette::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID FAR* ppvObj);
```

Determines if the DirectDrawPalette object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by DirectDraw.

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

riid
Reference identifier of the interface being requested.

ppvObj
Address of a pointer that receives the interface pointer if the request is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectDrawPalette::QueryInterface** method allows DirectDrawPalette objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirectDrawPalette::Release

```
ULONG Release();
```

Decreases the reference count of the DirectDrawPalette object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

The DirectDrawPalette object deallocates itself when its reference count reaches 0. Use the IDirectDrawPalette::AddRef method to increase the reference count of the object by 1.

IDirectDrawPalette::SetEntries

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

Changes entries in a DirectDrawPalette immediately. The palette must be attached to a surface using the IDirectDrawSurface::SetPalette method before **IDirectDrawPalette::SetEntries** can be used.

- 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 not used at this time and must be set to 0.

dwStartingEntry
Specifies the first entry to be set.

dwCount
Specifies the number of palette entries to be changed.

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

IDirectDrawClipper Interface Method Groups

Applications use the methods of the IDirectDraw interface to create DirectDraw objects and work with system-level variables. The methods can be organized into the following groups:

Allocating memory	Initialize
Clip lists	IsClipListChanged SetClipList SetHWnd
Handles	GetClipList GetHWnd
IUnknown	AddRef QueryInterface Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectDrawClipper object without affecting the functionality of the original interface.

IDirectDrawClipper::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectDrawClipper object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

When the DirectDraw object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirectDrawClipper::Release method to decrease the reference count of the object by 1.

IDirectDrawClipper::GetClipList

```
HRESULT GetClipList(LPRECT lpRect,
    LPRGNDATA lpClipList, LPDWORD lpdwSize);
```

Returns 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
Set by **IDirectDrawClipper::GetClipList** to indicate the size of the resulting clip list.

IDirectDrawClipper::GetHWnd

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

Returns the hWnd 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 hWnd previously associated with this DirectDrawClipper by the **IDirectDrawClipper::SetHWnd** method.

IDirectDrawClipper::Initialize

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

Initializes a DirectDrawClipper.

- Returns DDERR_ALREADYINITIALIZED.

lpDD

Address of the DirectDraw structure that represents the DirectDraw object.

dwFlags

This parameter is not used at this time and must be set to 0.

This method is provided for compliance with the Component Object Model (COM) protocol. Since the DirectDrawClipper object is initialized when it is created, calling this method will always result in the DDERR_ALREADYINITIALIZED return value.

IDirectDrawClipper::IsClipListChanged

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

Monitors the status of the clip list if an hWnd is associated with a DirectDrawClipper.

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

lpbChanged
Address of a Boolean value set to TRUE if the clip list has changed.

IDirectDrawClipper::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID FAR * ppvObj);
```

Determines if the DirectDrawClipper object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by DirectDraw.

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

riid
Reference identifier of the interface being requested.

ppvObj
Address of a pointer that receives the interface pointer if the request is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectDrawClipper::QueryInterface** method allows DirectDrawClipper objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirectDrawClipper::Release

```
ULONG Release();
```

Decreases the reference count of the DirectDrawClipper object by 1. This method is part of the IUnknown interface inherited by DirectDraw.

- Returns the new reference count of the object.

The DirectDrawClipper object deallocates itself when its reference count reaches 0. Use the IDirectDrawClipper::AddRef method to increase the reference count of the object by 1.

IDirectDrawClipper::SetClipList

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

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

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

  DDERR_INVALIDOBJECT
  DDERR_CLIPPERISUSINGHWND
  DDERR_INVALIDPARAMS
  DDERR_INVALIDCLIPLIST
  DDERR_OUTOFMEMORY

lpClipList

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

dwFlags

This parameter is not used at this time and must be set to 0.

The clip list is a series of rectangles that describes the visible areas of the surface. The clip list cannot be set if an hWnd is already associated with the DirectDrawClipper object. Note that the IDirectDrawSurface::BltFast method cannot clip.

IDirectDrawClipper::SetHWnd

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

Sets the hWnd that will obtain the clipping information.

- Returns DD_OK if successful, or one of the following error values otherwise:
  DDERR_INVALIDOBJECT DDERR_INVALIDPARAMS
  DDERR_INVALIDCLIPLIST DDERR_OUTOFMEMORY

dwFlags
This parameter is not used at this time and must be set to 0.

hWnd
The hWnd that obtains the clipping information.

DDBLTBATCH

```
typedef struct _DDBLTBATCH{
    LPRECT              lprDest;
    LPDIRECTDRAWSURFACE lpDDSSrc;
    LPRECT              lprSrc;
    DWORD               dwFlags;
    LPDDBLTFX           lpDDBltFx;
} DDBLTBATCH,FAR *LPDDBLTBATCH;
```

Passes blit operations to the IDirectDrawSurface::BltBatch method.

lprDest
Address of a RECT structure that defines the destination for the blit.

lpDDSSrc
Address of a DirectDrawSurface that will be the source of the blit.

lprSrc
Address of a RECT structure that defines the source rectangle of the blit.

dwFlags
Specifies the optional control flags.

DDBLT_ALPHADEST
Uses either the alpha information in the 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 in the DDBLTFX structure as the alpha channel for the destination surface for this blit.

DDBLT_ALPHADESTNEG
The NEG suffix indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHADESTSURFACEOVERRIDE
Uses the **lpDDSAlphaDest** member in the **DDBLTFX** structure as the alpha channel for the destination surface for this blit.

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

DDBLT_ALPHASRC
Uses either the alpha information in the 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 in the **DDBLTFX** structure as the alpha channel for the source for this blit.

DDBLT_ALPHASRCNEG
The NEG suffix indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHASRCSURFACEOVERRIDE
Uses the **lpDDSAlphaSrc** member in the **DDBLTFX** structure as the alpha channel for the 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 in the DDBLTFX structure as the RGB color with which to fill the destination rectangle on the destination surface.

DDBLT_DDFX

Uses the **dwDDFX** member in the **DDBLTFX** structure to specify the effects to be used for this blit.

DDBLT_DDROPS

Uses the **dwDDROPS** member in the **DDBLTFX** structure to specify the raster operations 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 in 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 in the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP

Uses the **dwROP** member in the **DDBLTFX** structure for the raster operation for this blit. The raster operations are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE

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

DDBLT_ZBUFFER

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

DDBLT_ZBUFFERDESTCONSTOVERRIDE

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

DDBLT_ZBUFFERDESTOVERRIDE

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

DDBLT_ZBUFFERSRCCONSTOVERRIDE

A z-buffered blit using the **dwZSrcConst** and **dwZBufferOpCode** members in 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 in 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 IDirectDrawSurface::Blt method. It is also part of the DDBLTBATCH structure used with the IDirectDrawSurface::BltBatch method.

dwSize

Size of the structure. This must be initialized before the structure is used.

dwDDFX

Specifies the 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

Specifies the Win32 raster operations.

dwDDROP

Specifies the 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.

dwAlphaEdgeBlend

Alpha used for edge blending.

dwAlphaEdgeBlendBitDepth

Bit depth of the constant for an alpha edge blend.

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. It contains a

DDSCAPS structure used in this context to describe what kinds of DirectDrawSurfaces can be created. It may not be possible to simultaneously create all of the surfaces described by these capabilities.

dwSize

Size of the structure. This must be initialized before the structure is used.

dwCaps

Specifies driver-specific capabilities.

DDCAPS_3D

Indicates the display hardware has 3D acceleration.

DDCAPS_ALIGNBOUNDARYDEST

Indicates that DirectDraw will support only those source rectangles with the x-axis aligned to the DDCAPS.dwAlignBoundaryDest boundaries of the surface.

DDCAPS_ALIGNBOUNDARYSRC

Indicates that DirectDraw will support only those source rectangles with the x-axis aligned to the DDCAPS.dwAlignBoundarySrc boundaries of the surface.

DDCAPS_ALIGNSIZEDEST

Indicates that DirectDraw will support only those source rectangles whose x-axis size, in bytes, are DDCAPS.dwAlignSizeDest multiples.

DDCAPS_ALIGNSIZESRC

Indicates that DirectDraw will support only those source rectangles whose x-axis size, in bytes, are DDCAPS.dwAlignSizeSrc multiples.

DDCAPS_ALIGNSTRIDE

Indicates that DirectDraw will create display memory surfaces that have a stride alignment equal to the DDCAPS.dwAlignStrideAlign value.

DDCAPS_ALPHA

Indicates the display hardware supports an alpha channel during blit operations.

DDCAPS_BANKSWITCHED

Indicates the display hardware is bank switched, and is potentially very slow at random access to VRAM.

DDCAPS_BLT

Indicates that display hardware is capable of blit operations.

DDCAPS_BLTCOLORFILL

Indicates that display hardware is capable of color filling with blitter.

DDCAPS_BLTDEPTHFILL

Indicates that display hardware is capable of depth filling z-buffers with 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 the color key is hardware assisted.

DDCAPS_GDI

Indicates that display hardware is shared with GDI.

DDCAPS_NOHARDWARE

Indicates 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 scan line.

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 IDirectDrawSurface::UpdateOverlayZOrder as a z-value for overlays to control their layering.

dwCaps2

Specifies more driver-specific capabilities.

DDCAPS2_CERTIFIED

Indicates that display hardware is certified.

DDCAPS2_NO2DDURING3DSCENE

Indicates that 2D operations such as IDirectDrawSurface::Blt and IDirectDrawSurface::Lock cannot be performed on any surfaces that Direct3D is using between IDirect3DDevice::BeginScene and IDirect3DDevice::EndScene method calls.

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 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

Specifies 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 only works 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 only valid for blit operations.

DDFXCAPS_BLTSHRINKXN

Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is only valid for blit operations.

DDFXCAPS_BLTSHRINKY

Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is only valid for blit operations.

DDFXCAPS_BLTSHRINKYN

Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is only valid for blit operations.

DDFXCAPS_BLTSTRETCHX

Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is only valid for blit operations.

DDFXCAPS_BLTSTRETCHXN

Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is only valid for blit operations.

DDFXCAPS_BLTSTRETCHY

Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is only valid for blit operations.

DDFXCAPS_BLTSTRETCHYN

Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is only valid 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 only works 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 only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKXN

Supports integer shrinking ($\times 1$, $\times 2$, and so on) of a surface along the x-axis (horizontally). This flag is only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates 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 only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKYN

Supports integer shrinking (x1, x2, and so on) of a surface along the y-axis (vertically). This flag is only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates 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 only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHXN

Supports integer stretching (x1, x2, and so on) of a surface along the x-axis (horizontally). This flag is only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates 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 only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHYN

Supports integer stretching (x1, x2, and so on) of a surface along the y-axis (vertically). This flag is only valid for DDSCAPS_OVERLAY surfaces. This flag only indicates the capabilities of a surface; it does not indicate that stretching is available.

dwFXAlphaCaps

Specifies 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 only be set 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 the depth of the alpha channel data can be 1, 2, 4, or 8. The NEG suffix indicates this alpha channel 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 only be set 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 only be set 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 the depth of the alpha channel data can be 1, 2, 4, or 8. The NEG suffix indicates this alpha channel 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 only be set if DDFXCAPS_ALPHASURFACES has been set. Used for overlays.

dwPalCaps

Specifies palette capabilities.

DDPCAPS_1BIT

The index is 1 bit. There are two entries in the palette table.

DDPCAPS_2BIT

The index is 2 bits. There are four entries in the palette table.

DDPCAPS_4BIT

The index is 4 bits. There are sixteen entries in the palette table.

DDPCAPS_8BIT

The index is 8 bits. There are 256 entries in the palette table.

DDPCAPS_8BITENTRIES

An index to an 8-bit color index. This field is only valid 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 one byte long and is an index to an 8-bpp palette on the destination surface.

DDPCAPS_ALLOW256

Indicates this palette can have all 256 entries defined.

DDPCAPS_PRIMARYSURFACE

Indicates the palette is attached to the primary surface. Changing this table has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_PRIMARYSURFACELEFT

Indicates the palette is attached to the primary surface on the left. Changing this table has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_VSYNC

Indicates the palette can be modified synchronously with the monitor's refresh rate.

dwSVCaps

Specifies stereo vision capabilities.

DDSVCAPS_ENIGMA

Indicates the stereo view is accomplished using Enigma encoding.

DDSVCAPS_FLICKER

Indicates the stereo view is accomplished using high-frequency flickering.

DDSVCAPS_REDBLUE

Indicates the stereo view is accomplished using red and blue filters applied to the left and right eyes. All images must adapt their color spaces for this process.

DDSVCAPS_SPLIT

Indicates the stereo view is accomplished with split-screen technology.

dwAlphaBltConstBitDepths

DDBD_2,4,8

dwAlphaBltPixelBitDepths

DDBD_1,2,4,8

dwAlphaBltSurfaceBitDepths

DDBD_1,2,4,8

dwAlphaOverlayConstBitDepths

DDBD_2,4,8

dwAlphaOverlayPixelBitDepths

DDBD_1,2,4,8

dwAlphaOverlaySurfaceBitDepths

DDBD_1,2,4,8

dwZBufferBitDepths

DDBD_8,16,24,32

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

Indicates a DDSCAPS structure with general capabilities.

dwMinOverlayStretch

Minimum overlay stretch factor multiplied by 1000. For example, 1.0 = 1000, 1.3 = 1300.

dwMaxOverlayStretch

Maximum overlay stretch factor multiplied by 1000. For example, 1.0 = 1000, 1.3 = 1300.

dwMinLiveVideoStretch

Minimum live video stretch factor multiplied by 1000. For example, 1.0 = 1000, 1.3 = 1300.

dwMaxLiveVideoStretch

Maximum live video stretch factor multiplied by 1000. For example, 1.0 = 1000, 1.3 = 1300.

dwMinHwCodecStretch

Minimum hardware codec stretch factor multiplied by 1000. For example 1.0 = 1000, 1.3 = 1300.

dwMaxHwCodecStretch

Maximum hardware codec stretch factor multiplied by 1000. For example, 1.0 = 1000, 1.3 = 1300.

dwReserved1,**dwReserved2**,**dwReserved3**

Reserved for future use.

dwSVBCaps

Specifies the driver-specific capabilities for system memory to display memory blits.

dwSVBCKeyCaps

Specifies the driver color key capabilities for system memory to display memory blits.

dwSVBFXCaps

Specifies the driver FX capabilities for system memory to display memory blits.

dwSVBRops[DD_ROP_SPACE]

Specifies the raster operations supported for system memory to display memory blits.

dwVSBCaps

Specifies the driver-specific capabilities for display memory to system memory blits.

dwVSBCKeyCaps

Specifies the driver color key capabilities for display memory to system memory blits.

dwVSBFXCaps

Specifies the 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

Specifies the driver-specific capabilities for system memory to system memory blits.

dwSSBCKeyCaps

Specifies the driver color key capabilities for system memory to system memory blits.

dwSSBCFXCaps

Specifies the 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**, **dwReserved6**

Reserved for future use.

The values for the **dw...BitDepths** members (**dwAlphaBltConstBitDepths**, **dwAlphaBltPixelBitDepths**, **dwAlphBltSurfaceBitDepths**, and so on) are:

DDBD_1	1 bit per pixel.
DDBD_2	2 bits per pixel.
DDBD_4	4 bits per pixel.
DDBD_8	8 bits per pixel.
DDBD_16	16 bits per pixel.
DDBD_24	24 bits per pixel.
DDBD_32	32 bits per pixel.

DDCOLORKEY

```
typedef struct _DDCOLORKEY{
    DWORD  dwColorSpaceLowValue;
    DWORD  dwColorSpaceHighValue;
} DDCOLORKEY,FAR* LPDDCOLORKEY;
```

Describes a source or destination color key or color space. A color key is specified if the low and high range values are the same.

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 IDirectDrawSurface::UpdateOverlay method.

dwSize
Size of the structure. This 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.I
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 not used at this time 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.

dwSize
Size of the structure. This must be initialized before the structure is used.

dwFlags
Specifies the 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
The surface is 1-bit color indexed.

DDPF_PALETTEINDEXED2
The surface is 2-bit color indexed.

DDPF_PALETTEINDEXED4
The surface is 4-bit color indexed.

DDPF_PALETTEINDEXED8

The surface is 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 (DDBD_4,8,16,24,32)

dwYUVBitCount

YUV bits per pixel (DDBD_4,8,16,24,32)

dwZBufferBitDepth

Z-buffer bit depth. (DDBD_8,16,24,32)

dwAlphaBitDepth

Alpha channel bit depth. (DDBD_1,2,4,8)

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. It is part of the DDCAPS structure that is used to describe the capabilities of the DirectDraw object.

dwCaps

Specifies the capabilities of the surface.

DDSCAPS_3D

Supported for backwards 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 have off-screen surfaces allocated from the correct heap. If this flag is set for a heap, the surface will not be allocated from that heap.

DDSCAPS_ALLOCONLOAD

Indicates that memory for the surface is not allocated until the surface is loaded using the IDirect3DTexture::Load method.

DDSCAPS_ALPHA

Indicates that this surface contains alpha information. The pixel format must be interrogated 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. Generally, this capability is set by the IDirectDraw::CreateSurface method when the DDSCAPS_FLIP capability is set. Only the surface that immediately precedes the DDSCAPS_FRONTBUFFER 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 **IDirectDraw::CreateSurface** method, a standalone back buffer is being created. This surface could be attached to a front buffer, another back buffer, or both to form a flipping surface structure after this method is called. See the IDirectDrawSurface::AddAttachedSurface method for a detailed description of the behaviors in this case. DirectDraw supports *n* surfaces in a surface flipping structure.

DDSCAPS_COMPLEX

Indicates a complex surface structure is being described. A complex surface structure results in the creation of more than one surface. The additional surfaces are attached to the root surface. The complex structure can only be destroyed 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 **IDirectDraw::CreateSurface** method, the DDSCAPS_FRONTBUFFER and DDSCAPS_BACKBUFFER capabilities are not set. Both can be set by this method on resulting creations. The **dwBackBufferCount** member in 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 through the **IDirectDraw::CreateSurface** method.

DDSCAPS_FRONTBUFFER

Indicates that this surface is the front buffer of a surface flipping structure. It is generally set by the IDirectDraw::CreateSurface method when the DDSCAPS_FLIP capability is set. If this capability is sent to the **IDirectDraw::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 on resulting creations. See the

IDirectDrawSurface::AddAttachedSurface method for a detailed description of the behaviors in this case.

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 with the **IDirectDrawSurface::AddAttachedSurface** method, or implicitly by the **IDirectDraw::CreateSurface** method. If this capability is set, then DDSCAPS_TEXTURE must also be set.

DDSCAPS_MODEX

Indicates that this surface is a 320 x 200 or 320 x 240 ModeX 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 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 if 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 and is set 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 protection 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;
```

Is passed to the IDirectDraw::CreateSurface method with a description of the surface that should be created. The relevant members differ for each potential type of surface.

dwSize
Size of the structure. This must be initialized before the structure is used.

dwFlags
Specifies the optional control flags.

DDSD_ALL
All input members are valid.

DDSD_ALPHABITDEPTH
Indicates the **dwAlphaBitDepth** member is valid.

DDSD_BACKBUFFERCOUNT
Indicates the **dwBackBufferCount** member is valid.

DDSD_CAPS
Indicates the **ddsCaps** member is valid.

DDSD_CKDESTBLT
Indicates the **ddckCKDestBlt** member is valid.

DDSD_CKDESTOVERLAY
Indicates the **ddckCKDestOverlay** member is valid.

DDSD_CKSRCBLT
Indicates the **ddckCKSrcBlt** member is valid.

DDSD_CKSRCOVERLAY
Indicates the **ddckCKSrcOverlay** member is valid.

DDSD_HEIGHT
Indicates the **dwHeight** member is valid.

DDSD_LPSURFACE
Indicates the **lpSurface** member is valid.

DDSD_MIPMAPCOUNT
Indicates the **dwMipMapCount** member is valid.
DDSD_PITCH
Indicates the **lPitch** member is valid.
DDSD_PIXELFORMAT
Indicates the **ddpfPixelFormat** member is valid.
DDSD_REFRESHRATE
Indicates the **dwRefreshRate** member is valid.
DDSD_WIDTH
Indicates the **dwWidth** member is valid.
DDSD_ZBUFFERBITDEPTH
Indicates 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 IDirectDraw, IDirectDrawSurface, IDirectDrawPalette, and IDirectDrawClipper methods. For a list of the error codes each method is capable of returning, see the individual method descriptions.

DD_OK

The request completed successfully.

DDERR_ALREADYINITIALIZED

The object has already been initialized.

DDERR_BLTFASTCANTCLIP

A clipper object is attached to a source surface that has passed into a call to the IDirectDrawSurface::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 clipper object that is already monitoring an hWnd.

DDERR_COLORKEYNOTSET

No source color key is specified for this operation.

DDERR_CURRENTLYNOTAVAIL

No support is currently available.

DDERR_DCALREADYCREATED

A device context has already been returned for this surface. Only one device context 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 CooperativeLevel HWND 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 CooperativeLevel HWND 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 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 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 clipper 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 IDirectDraw::SetCooperativeLevel method being called.

DDERR_NODC

No device context (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 an HWND, or no HWND has been previously set as the CooperativeLevel HWND.

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 IDirectDrawSurface::GetOverlayPosition method is called on an overlay that the IDirectDrawSurface::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 is not using a 4-bit color palette and the requested operation requires a 4-bit color palette.

DDERR_NOT4BITCOLORINDEX

The DirectDrawSurface is not using a 4-bit color index palette and the requested operation requires a 4-bit color index palette.

DDERR_NOT8BITCOLOR

The DirectDrawSurface 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 invoke 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 cannot be carried out because there is no hardware support for z-buffers when creating a z-buffer in display memory or when performing a z-aware blit.

DDERR_NOZOVERLAYHW

The overlay surfaces cannot be z-layered based on their *BltOrder* because the hardware does not support z-layering 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 IDirectDrawSurface::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 this 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 this surface is refused because the surface memory is gone. The DirectDrawSurface object representing this surface should have the IDirectDrawSurface::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.

About DirectSound

The Microsoft® DirectSound™ application programming interface (API) is the audio component of the Microsoft Windows® 95 DirectX™ 2 Software Development Kit (SDK) that provides low-latency mixing, hardware acceleration, and direct access to the sound device. DirectSound provides this functionality while maintaining compatibility with existing Windows 95-based applications and device drivers.

DirectX 2 allows you, as an application developer, access to the display and audio hardware while insulating you from the specific details of that hardware. The overriding design goal in DirectX 2 is speed. Instead of providing a high-level set of functions, DirectSound provides a device-independent interface, allowing applications to take full advantage of the capabilities of the audio hardware.

Object Types

The most fundamental type of object is the DirectSound object, which represents the sound card itself. The DirectSound object is controlled by the IDirectSound Component Object Model (COM) interface; the methods of this interface allow the application to change the characteristics of the card.

The second type of object is a sound buffer. DirectSound uses primary and secondary sound buffers. Primary sound buffers represent the audio data that is actually heard by the user, while secondary sound buffers represent individual source sounds. DirectSound provides controls for primary and secondary sound buffers in the IDirectSoundBuffer interface.

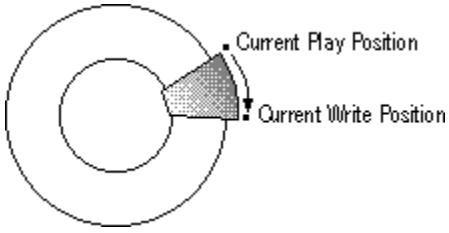
Primary buffers control sound characteristics, such as output format and total volume. Also, your application can write directly to the primary buffer. In this case, however, the DirectSound mixing and hardware acceleration features are not available. In addition, writing directly to the primary buffer may interfere with other DirectSound applications. When possible, your application should write to secondary buffers instead of the primary buffer. Secondary buffers allow the system to emulate features that might not be present in the hardware; they also allow an application to share the sound card with other applications in the system.

Secondary buffers represent single sound sources used by an application. Each buffer can be played or stopped independently. DirectSound mixes all playing buffers into the primary buffer, then outputs the primary buffer to the sound device. Secondary buffers can reside in hardware or system buffers; hardware buffers are mixed by the sound device without any system processing overhead.

Secondary sound buffers can be either static buffers or streaming buffers. A static buffer means that the buffer contains an entire sound. A streaming buffer means that the buffer only contains part of a sound, and, therefore, your application must continually write new data to the buffer while it is playing. DirectSound will attempt to store static buffers using sound memory located on the sound hardware, if available. Buffers stored on the sound hardware do not consume system processing time when they are played because the mixing is done in the hardware. Reusable sounds, such as gunshots, are the perfect candidates for static buffers.

Your applications will work with two significant positions within a sound buffer: the current play position and the current write position. The current play position indicates the location in the buffer where the sound is being played. The current write position indicates the location where you can safely change the data in the buffer.

The following illustration shows the relationship between the current play and current write positions.



Although DirectSound buffers are conceptually circular, they are implemented using contiguous, linear memory. When the current play position reaches the end of the buffer, it wraps back to the beginning of the buffer.

The DirectSound Object

Each sound device installed in the system is represented by a DirectSound object that is accessed through the IDirectSound interface. Your application can create a DirectSound object by calling the DirectSoundCreate function that returns an **IDirectSound** interface. DirectSound objects installed in the system can be enumerated by calling the DirectSoundEnumerate function.

Windows is a multitasking operating system. Typically, users run several programs at once and expect them all to share resources. DirectSound objects share sound devices by tracking the input focus, only producing sound when their owning application has the input focus. When an application loses the input focus, the audio streams from that object are muted. Multiple applications can create DirectSound objects for the same sound device. When the input focus changes between applications, the audio output automatically switches from one application's streams to another's. In this way, applications do not have to repeatedly play and stop their buffers when the input focus changes.

Note The header file for DirectSound includes C programming macro definitions for the methods of the **IDirectSound** and IDirectSoundBuffer interfaces.

The DirectSoundBuffer Object

Each sound or audio stream is represented by a DirectSoundBuffer object that your application can access through the IDirectSoundBuffer interface. DirectSoundBuffer objects can be created by calling the IDirectSound::CreateSoundBuffer method that returns an **IDirectSoundBuffer** interface.

Applications are also able to create primary or secondary sound buffers. As previously stated, a secondary sound buffer represents a single sound or audio stream. A primary buffer represents the output audio stream that can be a composite of several mixed secondary buffers. In the current implementation, each DirectSound object has only one primary buffer.

Your application can write data into sound buffers by locking the buffer, writing data to the buffer, and unlocking the buffer. A buffer can be locked by calling the IDirectSoundBuffer::Lock method. This method returns a pointer to the locked portion of the buffer. Once there, it is possible to copy audio data to the buffer. After writing data to the buffer, you must unlock the buffer and complete the write operation by calling the IDirectSoundBuffer::Unlock method.

The primary sound buffer contains the data that is heard. You can play audio data from a secondary sound buffer by using the IDirectSoundBuffer::Play method. This method causes DirectSound to begin mixing the secondary buffer into the primary buffer. By default, **IDirectSoundBuffer::Play** plays the buffer once and stops at the end. You can also play a sound repeatedly in a continuous loop by specifying the DSBPLAY_LOOPING flag when calling this method. A buffer that is playing can be stopped by using the IDirectSoundBuffer::Stop method.

Generally, the duration of a sound determines how your application uses the associated sound buffer. If the sound data is only a few seconds long, you can use a static buffer to store the sound. If the sound is longer than that, you should use a streaming buffer.

Your application can create a DirectSoundBuffer object that has a static buffer by using the IDirectSound::CreateSoundBuffer method and specifying the DSBCAPS_STATIC flag. DirectSound attempts to store static buffers using sound memory located on the sound hardware, if that memory is available. Buffers stored on sound hardware do not consume system processing time when they are played because the mixing is done in the hardware. Reusable sounds, such as engine roars, cheers, and jeers, are perfect candidates for static buffers.

Streaming buffers can also use hardware mixing if it is supported by the sound device; however, this is efficient only when your application runs on computers with fast data buses, such as the peripheral component interconnect (PCI) bus. If the computer does not have a fast bus, the data transfer overhead outweighs the benefits of hardware mixing. DirectSound will locate streaming buffers in hardware only if the sound device is located on a fast bus.

Note The DSBCAPS_STATIC flag used with the **IDirectSound::CreateSoundBuffer** method determines whether the buffer is static or streaming. If this flag is specified, DirectSound creates a static buffer; otherwise, DirectSound creates a streaming buffer.

Software Emulation

DirectSound can emulate in software the features that a particular sound card does not directly support without loss of functionality. Applications can query DirectSound to determine the capabilities of the audio hardware by using the IDirectSound::GetCaps method. A high-performance game, for example, can use this information to scale its audio features.

Device Drivers

DirectSound accesses the sound hardware through the DirectSound hardware-abstraction layer (HAL), an interface that is implemented by the audio-device driver. This is a Windows 95 audio-device driver that has been modified to support the HAL. This driver architecture provides backward compatibility with existing Windows-based applications. The DirectSound HAL provides the following functionality:

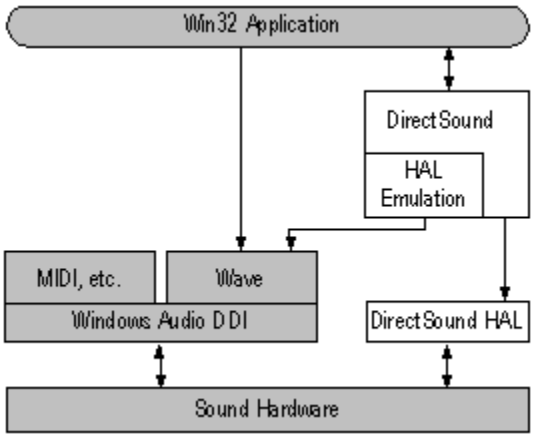
- Acquires and releases control of the audio hardware.
- Describes the capabilities of the audio hardware.
- Performs the specified operation when hardware is available.
- Fails the operation request when hardware is unavailable.

The device driver does not perform any software emulation; it simply reports the capabilities of the hardware to DirectSound and passes requests from DirectSound to the hardware. If the hardware cannot perform a requested operation, the device driver fails the request and DirectSound emulates the operation.

If a DirectSound driver is not available, DirectSound communicates with the audio hardware through the standard Windows 95 or Windows 3.1 audio-device driver. In this case, all DirectSound features are still available through software emulation, but hardware acceleration is not possible.

System Integration

The following diagram shows the relationships between DirectSound and other system audio components.



Using a device driver for the sound hardware that implements the DirectSound HAL provides the best performance for playing audio. The device driver implements each function of the HAL to leverage the architecture of the sound hardware and provide functionality and high performance. The HAL describes the capabilities of the hardware to DirectSound and passes requests from DirectSound to the hardware. If the hardware cannot handle the request, the driver will fail the call. DirectSound then emulates the request in software.

Your application can use DirectSound features even when no DirectSound driver is present. If the sound hardware does not have an installed DirectSound driver, DirectSound uses its HAL emulation layer. This layer uses the Windows multimedia waveform-audio functions.

The DirectSound functions and the waveform-audio functions provide alternative paths to the waveform-audio portion of the sound hardware. A single device provides access from one path at a time. If a waveform-audio driver has allocated a device, trying to allocate that same device using DirectSound will fail. Similarly, if a DirectSound driver has allocated a device, trying to allocate the device using the waveform-audio driver will fail.

If your application needs to use both sets of functions, you should use each set sequentially. For example, you could open the sound hardware by using the DirectSoundCreate function, play sounds using the IDirectSound and IDirectSoundBuffer interfaces, and close the sound hardware by using the IDirectSound::Release method. The sound hardware would then be available for the waveform-audio functions of the Microsoft Win32® SDK.

Also, if two sound devices are installed in the system, your application can access each device independently through either DirectSound or the waveform-audio functions.

The waveform-audio functions continue to be a practical solution for certain applications. For example, your application can easily play a single sound or audio stream, such as an introductory sound, by using the **PlaySound** function or the **WaveOut** function.

Note Microsoft Video For Windows currently uses the waveform-audio functions to output the audio track of an .avi file. Therefore, if your application is using DirectSound and you play an .avi file, the audio track will not be audible. Similarly, if you play an .avi file and attempt to create a DirectSound object, the creation function will return an error.

For now, applications can release the DirectSound object by calling the **IDirectSound::Release** method before playing an .avi file, and then re-create and reinitialize the DirectSound object and its DirectSoundBuffer objects when the video finishes playing. For more information about **IDirectSound::Release,** see IUnknown Interface.

Mixing

The most used feature of DirectSound is the low-latency mixing of audio streams. Your application can create one or more secondary sound buffers and write audio data to them. You can choose to play or stop any of these buffers. DirectSound mixes all playing buffers and writes the result to the primary sound buffer, which supplies the sound hardware with audio data. There are no limitations to the number of buffers that can be mixed, except the practicalities of available processing time.

Low-latency mixing allows the user to experience no perceptible delay between the time that a buffer plays and the time that the speakers reproduce the sound. In practical terms, this means that the latency is 20 milliseconds or less. The DirectSound mixer provides 20 milliseconds of latency, so there is no perceptible delay before play begins. Under these conditions, if your application plays a buffer and immediately begins a screen animation, the audio and video appear to start synchronously. However, if DirectSound must use the HAL emulation layer (if a DirectSound driver for the sound hardware is not present), the mixer cannot achieve low latency and a hardware-dependent delay, typically 100-150 milliseconds, occurs before the sound is reproduced.

Because only one application at a time can open a particular DirectSound device, only buffers from a single application are audible at any given instance.

Hardware Acceleration

DirectSound automatically takes advantage of accelerated sound hardware, including hardware mixing and hardware sound-buffer memory. Your application does not need to query the hardware or program specifically to use hardware acceleration.

However, for you to make the best possible use of the available hardware resources, you can query DirectSound to receive a full description of the hardware capabilities of the sound device. From this information, you can specify which sound buffers should receive hardware acceleration.

Because your application determines when to use each effect, when to play each sound buffer, and what priority each buffer should take, it can allocate hardware resources as it needs them.

Write Access to the Primary Buffer

The primary sound buffer outputs audio samples to the sound device. DirectSound provides direct write access to the primary buffer; however, this feature is useful for a very limited set of applications that require specialized mixing or other effects not supported by secondary buffers. Gaps in sound are difficult to avoid when an application writes directly to the primary buffer; those that access the primary buffer directly are subject to very stringent performance requirements.

A primary buffer is typically very small, so if your application writes directly to this kind of buffer, it must write blocks of data at short intervals in order to prevent the previous block in the buffer from repeating. During buffer creation, you cannot specify the size of the buffer and must accept the returned size once creation is complete.

When you obtain write access to a primary sound buffer, other DirectSound features become unavailable. Secondary buffers are not mixed and, consequently, hardware-acceleration mixing is unavailable. (When DirectSound mixes sounds from secondary buffers, it places the mixed audio data in the primary buffer.)

Most of your applications should use secondary buffers instead of directly accessing the primary buffer. Applications can write to a secondary buffer easily because the larger buffer size provides more time to write the next block of data, thereby minimizing the risk of gaps in the audio. Even if your application has simple audio requirements, such as using one stream of audio data that does not require mixing, it will achieve better performance by using a secondary buffer to play its audio data.

IDirectSound Interface

A DirectSound object describes the audio hardware on a system. The audio data itself resides in a buffer called a DirectSoundBuffer object. For more information about DirectSound buffers, see IDirectSoundBuffer Interface. The IDirectSound interface enables your application to define and control the sound card, speaker, and memory environment.

Device Capabilities

After calling the DirectSoundCreate function to create a DirectSound object, your application can retrieve the capabilities of the sound device by calling the IDirectSound::GetCaps method. For optimal performance, you should make this call to determine the capabilities of the resident sound card, then modify its sound parameters as appropriate.

Creating Buffers

After calling the DirectSoundCreate function to create a DirectSound object and investigating the capabilities of the sound device, your application can create and enumerate the sound buffers that contain audio data. The IDirectSound::CreateSoundBuffer method creates a sound buffer. The IDirectSound::DuplicateSoundBuffer method creates a second sound buffer using the same physical buffer memory as the first. If you duplicate a sound buffer, you can play both buffers independently without wasting buffer memory.

Your application must use the IDirectSound::SetCooperativeLevel method to set its cooperative level for a sound device before playing any sound buffers. Most applications use a standard priority level, DSSCL_NORMAL, which ensures that they will not conflict with other applications.

Speaker Configuration

The IDirectSound interface contains two methods that allow your application to investigate and set the configuration of the system's speakers. These methods are IDirectSound::GetSpeakerConfig and IDirectSound::SetSpeakerConfig. Currently recognized configurations include headphones, binaural headphones, stereo, quadraphonic, and surround sound.

Hardware Memory Management

Your application can use the IDirectSound::Compact method to move any onboard sound memory into a contiguous block to make the largest portion of free memory available.

IDirectSoundBuffer Interface

The IDirectSoundBuffer interface enables your application to work with buffers of audio data. Audio data resides in a DirectSound buffer. Your application creates DirectSound buffers for each sound or audio stream to be played.

The primary sound buffer represents the actual audio samples output to the sound device. These samples can be a single audio stream or the result of mixing several audio streams. The audio data in a primary sound buffer is typically not accessed directly by applications. However, the primary buffer can be used for control purposes, such as setting the output volume or wave format.

Secondary sound buffers represent a single output stream or sound. Your application can play these buffers into the primary sound buffer. Secondary sound buffers that play concurrently are mixed into the primary buffer, which is then sent to the sound device.

Play Management

Your application can use the IDirectSoundBuffer::Play and IDirectSoundBuffer::Stop methods to control the real-time playback of sound. You can also play a sound using **IDirectSoundBuffer::Play**. The buffer stops automatically when its end is reached. However, if looping is specified, the buffer repeats until **IDirectSoundBuffer::Stop** is called.

The IDirectSoundBuffer::Lock method retrieves a write pointer into the current sound buffer. After writing audio data into the buffer, you must unlock the buffer by using the IDirectSoundBuffer::Unlock method. You should not leave the buffer locked for extended periods.

To retrieve or set the current position in the sound buffer, call the IDirectSoundBuffer::GetCurrentPosition or IDirectSoundBuffer::SetCurrentPosition method.

Sound-Environment Management

To retrieve and set the volume at which a buffer is played, your application can use the IDirectSoundBuffer::GetVolume and IDirectSoundBuffer::SetVolume methods. Setting the volume on the primary sound buffer changes the waveform-audio volume of the sound card.

Similarly, by calling the IDirectSoundBuffer::GetFrequency and IDirectSoundBuffer::SetFrequency methods, you can retrieve and set the frequency at which audio samples are played. The frequency of the primary buffer cannot be changed.

To retrieve and set the pan, you can call the IDirectSoundBuffer::GetPan and IDirectSoundBuffer::SetPan methods. The pan of the primary buffer cannot be changed.

Retrieving Information

The IDirectSoundBuffer::GetCaps method retrieves the capabilities of the DirectSoundBuffer object. Your application can use the IDirectSoundBuffer::GetStatus method to determine if the current sound buffer is playing or if it has stopped.

You can use the IDirectSoundBuffer::GetFormat method to retrieve information about the format of the sound data in the buffer. The **IDirectSoundBuffer::GetFormat** and IDirectSoundBuffer::SetFormat methods can also be used to set the format of the sound data in the primary buffer.

Note Once a secondary buffer is created, its format is fixed. If a secondary buffer that uses another format is needed, you should create a new sound buffer with the necessary format.

Memory Management

Your application can use the IDirectSoundBuffer::Restore method to restore the sound buffer memory for a specified DirectSoundBuffer object. Although this is useful if the buffer has been lost, the **IDirectSoundBuffer::Restore** method restores only the memory itself. It cannot restore the content of the memory. Once the buffer memory is restored, it must be rewritten with valid sound data.

IUnknown Interface

Like all Component Object Model (COM) interfaces, the IDirectSound and IDirectSoundBuffer interfaces also include the **AddRef**, **Release**, and **QueryInterface** methods. The **AddRef** method increases the reference count of the object and the **Release** method decreases the reference count. Your applications can use the **QueryInterface** method to determine what additional interfaces an object supports.

When an object is created, its reference count is set to 1. Each time a new application binds to the object or a previously bound application binds to a different interface of the object, the reference count is increased by 1. Each time an application is released from an interface, the reference count is decreased by 1. The object deallocates itself when its reference count goes to 0. The **Release** method notifies the object that an application is no longer bound to the object.

Your application can use the **QueryInterface** method to ask an object if it supports a particular interface. If the interface is supported, it can be used immediately. If the interface is not needed, you must call the **Release** method to free it. **QueryInterface** allows objects to be extended by Microsoft and third parties without breaking or interfering with each other's existing or future functionality.

Note DirectSoundBuffer objects are owned by the DirectSound object that created them. When the DirectSound object is released, all buffers created by that object will be released as well and should not be referenced.

Implementation: A Broad Overview

This section describes the programming model for DirectSound and provides some guidelines for typical tasks.

Your application should follow these basic steps to implement DirectSound:

1. Create a DirectSound object by calling the DirectSoundCreate function.
2. Specify a cooperative level by calling the IDirectSound::SetCooperativeLevel method. Most applications use the lowest level, DSSCL_NORMAL.
3. Create secondary buffers using the IDirectSound::CreateSoundBuffer method. Your application need not specify that they are secondary buffers in the DSBUFFERDESC structure; creating secondary buffers is the default.
4. Load the secondary buffers with data. Use the IDirectSoundBuffer::Lock method to obtain a pointer to the data area and the IDirectSoundBuffer::Unlock method to set the data to the device.
5. Use the IDirectSoundBuffer::Play method to play the secondary buffers.
6. Stop all buffers when your application has finished playing sounds by using the IDirectSoundBuffer::Stop method of the DirectSoundBuffer object.
7. Release the secondary buffers.
8. Release the DirectSound object.

Your application can also perform the following optional operations:

- Set the output format of the primary buffer by creating a primary sound buffer and calling the IDirectSoundBuffer::SetFormat method. This operation requires your application to set the cooperative level to DSSCL_PRIORITY before setting the output format of the primary buffer.
- Create a primary sound buffer and play the buffer using the **IDirectSoundBuffer::Play** method. This guarantees that the primary buffer is always playing, even if no secondary buffers are playing. This action consumes some of the processing bandwidth, but it reduces startup time when the first secondary buffer is played.

Creating a DirectSound Object

The easiest way to create a DirectSound object is for your application to call the DirectSoundCreate function and specify a NULL GUID. The function will then attempt to create the object corresponding to the default window's wave device. You must then call the IDirectSound::SetCooperativeLevel method; no sound buffers can be played until this call has been made:

```
LPDIRECTSOUND lpDirectSound;
if(DS_OK == DirectSoundCreate(NULL, &lpDirectSound,
    NULL)) {
    // Create succeeded!
    lpDirectSound->lpVtbl->SetCooperativeLevel(lpDirectSound,
        hwnd, DSSCL_NORMAL);
    //  .
    //  .  Place code to access DirectSound object here.
    //  .
} else {
    // Create failed!
    //  .
    //  .
    //  .
}
```

The DirectSoundEnumerate function can be used to specify the particular sound device to create. To use this function, you must create a DSEnumCallback function and, in most cases, an instance data structure:

```
typedef struct {
    // storage for GUIDs
    // storage for device description strings
} APPINSTANCEDATA, *LPAPPINSTANCEDATA;
BOOL AppEnumCallbackFunction(
    LPGUID lpGuid,
    LPTSTR lpstrDescription,
    LPTSTR lpstrModule,
    LPVOID lpContext)
{
    LPAPPINSTANCEDATA lpInstance = (LPAPPINSTANCEDATA)
    lpContext;
    // Copy GUID into lpInstance structure.
    // Strcpy description string into lpInstance
    // structure.
    return TRUE; // Continue enumerating.
}
```

Then, to create the DirectSound object, you would use code like this:

```
AppInitDirectSound()
{
    APPINSTANCEDATA AppInstanceData;
    LPGUID lpGuid;
    LPDIRECTSOUND lpDirectSound;
    HRESULT hr;
    DirectSoundEnumerate(AppEnumCallbackFunction,
    &AppInstanceData);
    lpGuid = AppLetUserSelectDevice(&AppInstanceData);

    // The application should check the return value of
    // DirectSoundCreate for errors.

    hr = DirectSoundCreate(lpGuid, &lpDirectSound, NULL);
    //  .
```

```
    //   .
    //   .
}
```

The DirectSoundCreate function will fail if there is no sound device or if the sound device, as specified by the *lpGuid* parameter, has been allocated through the waveform-audio functions. You should prepare your applications for this call to fail so that they will either continue without sound or prompt the user to close the application that is using the sound device.

Creating a DirectSound Object Using CoCreateInstance

Use the following steps to create an instance of a DirectSound object using **CoCreateInstance**:

1 Initialize COM at the start of your application using CoInitialize(NULL).

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

2 Create your DirectSound object using **CoCreateInstance** and the IDirectSound::Initialize method rather than the DirectSoundCreate function.

```
dsrval = CoCreateInstance(&CLSID_DirectSound,
                                        NULL,
                                        CLSCTX_ALL,
                                        &IID_IDirectSound,
                                        &lpds);
if( !FAILED(dsrval) )
    dsrval = IDirectSound_Initialize( lpds, NULL);
```

CLSID_DirectSound is the class identifier of the DirectSound driver object class and *IID_IDirectSound* is the DirectSound interface that you should use. *lpds* is the uninitialized object **CoCreateInstance** returns.

Before you use the DirectSound object, you must call **IDirectSound::Initialize**. **IDirectSound::Initialize** takes the driver GUID parameter that **DirectSoundCreate** typically uses (NULL in this case). Once the DirectSound object is initialized, you can use and release the DirectSound object as if it had been created using **DirectSoundCreate**.

Before closing the application, shut down COM using **CoUninitialize**, as shown below.

```
CoUnitialize()
```

Querying the Hardware Capabilities

DirectSound allows your application to retrieve the hardware capabilities of the sound device being used by a DirectSound object. Most applications will not need to do this; DirectSound automatically takes advantage of hardware acceleration. However, high-performance applications can use this information to scale their sound requirements to the available hardware. For example, your application might play more sounds if hardware mixing is available.

To retrieve the hardware capabilities, use the IDirectSound::GetCaps method, which will fill in a DSCAPS structure:

```
AppDetermineHardwareCaps(LPDIRECTSOUND lpDirectSound)
{
    DSCAPS dscaps;
    HRESULT hr;
    dscaps.dwSize = sizeof(DSCAPS);
    hr = lpDirectSound->lpVtbl->GetCaps(lpDirectSound,
    &dscaps);
    if(DS_OK == hr) {
        // Succeeded, now parse DSCAPS structure.
        //  .
        //  .
        //  .
    }
    //  .
    //  .
    //  .
}
```

The **DSCAPS** structure contains information about the performance and resources of the sound device, including the maximum resources of each type and the resources that are currently available. There may be trade-offs between various resources; for example, allocating a single hardware streaming buffer might consume two static mixing channels. If your application scales to hardware capabilities, you should call the IDirectSound::GetCaps method between every buffer allocation to determine if there are enough resources for the next buffer creation.

Do not make assumptions about the behavior of the sound device, otherwise your application may work on some sound devices but not on others. Furthermore, advanced devices are under development that will behave differently than existing devices.

When allocating hardware resources, your application should attempt to allocate them as software buffers instead. Complete access to all hardware resources is not always available. For example, because Windows 95 is a multitasking operating system, the **IDirectSound::GetCaps** method might indicate a free resource, but by the time you attempt to allocate the resource, it may have been allocated to another application.

Creating a Basic Sound Buffer

To create a sound buffer, your application fills in a DSBUFFERDESC structure and then calls the IDirectSound::CreateSoundBuffer method. This creates a DirectSoundBuffer object and returns a pointer to an IDirectSoundBuffer interface. This interface can be used to write, manipulate, and play the buffer.

You should create buffers for the most important sounds first, and continue to create them in descending order of importance. DirectSound allocates hardware resources to the first buffer that can take advantage of them.

The following example illustrates how to create a basic secondary buffer:

```
BOOL AppCreateBasicBuffer(
    LPDIRECTSOUND lpDirectSound,
    LPDIRECTSOUNDBUFFER *lplpDsb)
{
    PCMWAVEFORMAT pcmwf;
    DSBUFFERDESC dsbdesc;
    HRESULT hr;
    // Set up wave format structure.
    memset(&pcmwf, 0, sizeof(PCMWAVEFORMAT));
    pcmwf.wf.wFormatTag = WAVE_FORMAT_PCM;
    pcmwf.wf.nChannels = 2;
    pcmwf.wf.nSamplesPerSec = 22050;
    pcmwf.wf.nBlockAlign = 4;
    pcmwf.wf.nAvgBytesPerSec =
        pcmwf.wf.nSamplesPerSec * pcmwf.wf.nBlockAlign;
    pcmwf.wBitsPerSample = 16;
    // Set up DSBUFFERDESC structure.
    memset(&dsbdesc, 0, sizeof(DSBUFFERDESC)); // Zero it out.
    dsbdesc.dwSize = sizeof(DSBUFFERDESC);
    dsbdesc.dwFlags = DSBCAPS_CTRLDEFAULT; // Need default controls
                                           (pan, volume, frequency).
    dsbdesc.dwBufferBytes = 3 * pcmwf.wf.nAvgBytesPerSec; // 3 second
                                                          buffer.
    dsbdesc.lpwfxFormat = (LPWAVEFORMATEX)&pcmwf;
    // Create buffer.
    hr = lpDirectSound->lpVtbl->CreateSoundBuffer(lpDirectSound,
        &dsbdesc, lplpDsb, NULL);
    if(DS_OK == hr) {
        // Succeeded! Valid interface is in *lplpDsb.
        return TRUE;
    } else {
        // Failed!
        *lplpDsb = NULL;
        return FALSE;
    }
}
```

Control Options

When creating a sound buffer, your application must specify the control options needed for that buffer. This can be done with the **dwFlags** member of the DSBUFFERDESC structure, which can contain one or more DSBCAPS_CTRL flags. DirectSound uses this information when allocating hardware resources to sound buffers. For example, a particular device might support hardware buffers but provide no pan control on those buffers. In this case, DirectSound would only use hardware acceleration if the DSBCAPS_CTRLPAN flag was not specified.

To obtain the best performance on all sound cards, your application should only specify those control options which it will actually use.

If your application calls a method that a buffer lacks, that method fails. For example, if you attempt to change the volume by using theIDirectSoundBuffer::SetVolume method, it will succeed if the DSBCAPS_CTRLVOLUME flag was specified when the buffer was created. Otherwise, the method fails and returns the DSERR_CONTROLUNAVAIL error code. Providing controls for the buffers helps to ensure that all applications run correctly on all existing or future sound devices.

Static and Streaming Sound Buffers

A static sound buffer contains a complete sound in memory. These buffers are convenient because the entire sound can be written once to the buffer.

A streaming buffer only represents a portion of a sound, such as a buffer that can hold three seconds of audio data that plays a two-minute sound. In this case, your application must periodically write new data into the sound buffer. However, a streaming buffer requires much less memory than a static buffer.

When creating a sound buffer, you can indicate that a buffer is static by specifying the DSBCAPS_STATIC flag. If this flag is not specified, the buffer is a streaming buffer.

If a sound device has onboard sound memory, DirectSound will attempt to place static buffers in the hardware memory. These buffers can then take advantage of hardware mixing, with the benefit that the processing system incurs little or no overhead to mix these sounds. This is particularly useful for sounds that will be played more than once, such as the sound a character makes while walking or a firearm going off, since the sound data only needs to be downloaded once to the hardware memory.

Streaming buffers are generally located in main system memory to allow efficient writing to the buffer, although hardware mixing can be used on peripheral component interconnect (PCI) machines or other fast buses. There are no requirements on the use of streaming buffers. For example, you can write an entire sound into a streaming buffer if it is big enough. In fact, if you do not intend to use the sound more than once, it may be more efficient to use a streaming buffer because there is no need for the sound data to be downloaded to the hardware memory.

Note The designation of a buffer as static or streaming is used by DirectSound to optimize performance; it does not restrict how you can use the buffer.

Hardware and Software Sound Buffers

A hardware sound buffer has its mixing performed by a hardware mixer located on the sound device. A software sound buffer has its mixing performed by the system central processing unit. In most cases, your application should simply specify whether the buffer is static or streaming; DirectSound will locate the buffer in hardware or software as appropriate.

If explicitly locating buffers in hardware or software is necessary, however, you can specify either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags in the DSBUFFERDESC structure. If the DSBCAPS_LOCHARDWARE flag is specified and there is insufficient hardware memory or mixing capacity, the buffer creation request will fail. Also, most existing sound devices do not have any hardware memory or mixing capacity, so no hardware buffers can be created on these devices.

The location of a sound buffer can be determined by calling the IDirectSoundBuffer::GetCaps method and checking the **dwFlags** member of the DSBCAPS structure for either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags. One or the other will always be specified.

Primary and Secondary Sound Buffers

Primary sound buffers represent the actual audio samples that the listener will hear. Secondary buffers each represent a single sound or stream of audio. Your application can create a primary buffer by specifying the DSBCAPS_PRIMARYBUFFER flag in the DSBUFFERDESC structure. If this flag is not specified, a secondary buffer will be created.

Usually you should create secondary sound buffers for each sound in a given application. Note that sound buffers can be reused by overwriting the old sound data with new data. DirectSound takes care of the hardware resource allocation and the mixing together of all buffers that are playing.

If your application uses secondary buffers, you may choose to create a primary sound buffer to perform certain controls. For example, you can control the hardware output format by calling the IDirectSoundBuffer::SetFormat method on the primary buffer. However, any methods that access the actual buffer memory, such as IDirectSoundBuffer::Lock and IDirectSoundBuffer::GetCurrentPosition, will fail.

If your application performs its own mixing, DirectSound provides write access to the primary buffer. You should write data into this buffer in a timely manner; if you do not update the data, the previous buffer will repeat itself, causing gaps in the audio. Write access to the primary buffer is only available if your application has the DSSCL_WRITEPRIMARY cooperative level. At this cooperative level, no secondary buffers can be played.

Note that primary sound buffers must be played with looping. Be sure that the DSBPLAY_LOOPING flag is set.

The following example shows how to obtain write access to the primary buffer:

```
BOOL AppCreateWritePrimaryBuffer(
    LPDIRECTSOUND lpDirectSound,
    LPDIRECTSOUNDBUFFER *lplpDsb,
    LPDWORD lpdwBufferSize,
    HWND hwnd)
{
    DSBUFFERDESC dsbdesc;
    DSBCAPS dsbcaps;
    HRESULT hr;
    // Set up wave format structure.
    memset(&pcmwf, 0, sizeof(PCMWAVEFORMAT));
    pcmwf.wf.wFormatTag = WAVE_FORMAT_PCM;
    pcmwf.wf.nChannels = 2;
    pcmwf.wf.nSamplesPerSec = 22050;
    pcmwf.wf.nBlockAlign = 4;
    pcmwf.wf.nAvgBytesPerSec =
        pcmwf.wf.nSamplesPerSec * pcmwf.wf.nBlockAlign;
    pcmwf.wBitsPerSample = 16;
    // Set up DSBUFFERDESC structure.
    memset(&lplpDsb, 0, sizeof(DSBUFFERDESC)); // Zero it out.
    dsbdesc.dwSize = sizeof(DSBUFFERDESC);
    dsbdesc.dwFlags = DSBCAPS_PRIMARYBUFFER;
    dsbdesc.dwBufferBytes = 0; // Buffer size is determined
                               // by sound hardware.
    dsbdesc.lpwfxFormat = NULL; // Must be NULL for primary buffers.

    // Obtain write-primary cooperative level.
    hr = lpDirectSound->lpVtbl->SetCooperativeLevel(lpDirectSound,
        hwnd, DSSCL_WRITEPRIMARY);
    if(DS_OK == hr) {
        // Succeeded! Try to create buffer.
        hr = lpDirectSound->lpVtbl->CreateSoundBuffer(lpDirectSound,
            &dsbdesc, lplpDsb, NULL);
        if(DS_OK == hr) {
            // Succeeded! Set primary buffer to desired format.
```

```
            hr = (*lplpDsb)->lpVtbl->SetFormat(*lplpDsb, &pcmwf);
            if(DS_OK == hr) {
                // If you want to know the buffer size, call GetCaps.
        dsbcaps.dwSize = sizeof(DSBCAPS);
                (*lplpDsb)->lpVtbl->GetCaps(*lplpDsb, &dsbcaps);
                *lpdwBufferSize = dsbcaps.dwBufferBytes;
                return TRUE;
            }
        }
    }
    // If we got here, then we failed SetCooperativeLevel.
    // CreateSoundBuffer, or SetFormat.
    *lplpDsb = NULL;
    *lpdwBufferSize = 0;
    return FALSE;
}
```

Writing to Sound Buffers

Your application can obtain write access to a sound buffer by using the IDirectSoundBuffer::Lock method. Once the sound buffer (memory) is locked, you can write or copy data to the buffer. The buffer memory must then be unlocked by calling the IDirectSoundBuffer::Unlock method.

Since streaming sound buffers usually wrap around and continue playing from the beginning of the buffer, DirectSound returns two write pointers when locking a sound buffer. For example, if you try to lock 300 bytes beginning at the midpoint of a 400 byte buffer, **IDirectSoundBuffer::Lock** would return a pointer to the last 200 bytes of the buffer, and a second pointer to the first 100 bytes. Depending on the offset and the length of the buffer, the second pointer may be NULL.

You should be aware that memory for a sound buffer can be lost in certain situations. In particular, this might occur when buffers are located in the hardware sound memory. In the most dramatic case, the sound card itself might be removed from the system while being used; this situation can occur with PCMCIA sound cards. It can also occur when an application with the write-primary cooperative level (DSSCL_WRITEPRIMARY flag) gains the input focus. If this flag is set, DirectSound makes all other sound buffers become lost so that the application with the focus can write directly to the primary buffer. If this happens, DirectSound returns the DSERR_BUFFERLOST error code in response to the **IDirectSoundBuffer::Lock** and IDirectSoundBuffer::Play methods. When the application lowers its cooperative level from write-primary, or loses the input focus, other applications can attempt to reallocate the buffer memory by calling the IDirectSoundBuffer::Restore method. If successful, this method restores the buffer memory and all other settings for the buffer, such as volume and pan settings. However, a restored buffer does not contain valid sound data. The owning application must rewrite the data to the restored buffer.

The following function writes data to a sound buffer using the IDirectSoundBuffer::Lock and IDirectSoundBuffer::Unlock methods:

```
BOOL AppWriteDataToBuffer(
    LPDIRECTSOUNDBUFFER lpDsb,
    DWORD dwOffset,
    LPBYTE lpbSoundData,
    DWORD dwSoundBytes)
{
    LPVOID lpvPtr1;
    DWORD dwBytes1;
    LPVOID lpvPtr2;
    DWORD dwBytes2;
    HRESULT hr;
    // Obtain write pointer.
    hr = lpDsb->lpVtbl->Lock(lpDsb, dwOffset, dwSoundBytes, &lpvPtr1,
        &dwBytes1, &lpvPtr2, &dwBytes2, 0);

    // If we got DSERR_BUFFERLOST, restore and retry lock.
    if(DSERR_BUFFERLOST == hr) {
        lpDsb->lpVtbl->Restore(lpDsb);
        hr = lpDsb->lpVtbl->Lock(lpDsb, dwOffset, dwSoundBytes,
            &lpvPtr1, &dwAudio1, &lpvPtr2, &dwAudio2, 0);
    }
    if(DS_OK == hr) {
        // Write to pointers.
        CopyMemory(lpvPtr1, lpbSoundData, dwBytes1);
        if(NULL != lpvPtr2) {
            CopyMemory(lpvPtr2, lpbSoundData+dwBytes1, dwBytes2);
        }
        // Release the data back to DirectSound.
        hr = lpDsb->lpVtbl->Unlock(lpDsb, lpvPtr1, dwBytes1, lpvPtr2,
            dwBytes2);
        if(DS_OK == hr) {
            // Success!
            return TRUE;
```

```
        }
    }
    // If we got here, then we failed Lock, Unlock, or Restore.
    return FALSE;
}
```

Using the DirectSound Mixer

It is easy to mix multiple streams with DirectSound. Your application can simply create secondary buffers, receiving an IDirectSoundBuffer interface for each sound. You can then use these interfaces to write data into the buffers using the IDirectSoundBuffer::Lock and IDirectSoundBuffer::Unlock methods and play the buffers using the IDirectSoundBuffer::Play method. The buffers can be stopped at any time by calling the IDirectSoundBuffer::Stop method.

The **IDirectSoundBuffer::Play** method will always start playing at the buffer's current position. The current position is specified by an offset, in bytes, into the buffer. The current position of a newly created buffer is 0. When a buffer is stopped, the current position immediately follows the next sample played. The current position can be set explicitly by calling the IDirectSoundBuffer::SetCurrentPosition method, and can be queried by calling the IDirectSoundBuffer::GetCurrentPosition method.

By default, **IDirectSoundBuffer::Play** will stop playing when it reaches the end of the buffer. This is the correct behavior for non-looping static buffers. (The current position will be reset to the beginning of the buffer at this point.) For streaming buffers or for static buffers that continuously repeat, your application should call **IDirectSoundBuffer::Play** and specify the DSBPLAY_LOOPING flag in the *dwFlags* parameter. This will cause the buffer to loop back to the beginning once it reaches the end.

For streaming buffers, your application is responsible for ensuring that the next block of data is written into the buffer before the play cursor loops back to the beginning. This can be done by using the Win32 functions **SetTimer** or **SetEvent** to cause a message or callback function to occur at regular intervals. In addition, many DirectSound applications will already have a real-time DirectDraw component which needs to service the display at regular intervals; this component should be able to service DirectSound buffers as well. For optimal efficiency, all applications should write at least one second ahead of the current play cursor to minimize the possibility of gaps in the audio output during playback.

The DirectSound mixer can obtain the best usage out of hardware acceleration if your application correctly specifies the DSBCAPS_STATIC flag for static buffers. This flag should be specified for any static buffers that will be reused. DirectSound will download these buffers to the sound hardware memory, where available, and will thereby not incur any processing overhead in mixing these buffers. The most important static buffers should be created first, in order to give them first priority for hardware acceleration.

The DirectSound mixer will produce the best sound quality if all of your application's sounds use the same wave format and the hardware output format is matched to the format of the sounds. If this is done, the mixer need not perform any format conversion.

The hardware output format can be changed by creating a primary buffer and calling the IDirectSoundBuffer::SetFormat method. Note that this primary buffer is for control purposes only; only applications with a cooperative level of DSSCL_PRIORITY or higher can call this function. DirectSound will then restore the hardware format to the format specified in the last **IDirectSoundBuffer::SetFormat** method call each time the application gains the input focus.

Using a Custom Mixer

Most applications should use the DirectSound mixer; it should be sufficient for almost all mixing needs and it automatically takes advantage of hardware acceleration, if available. However, if an application requires some other functionality that DirectSound does not provide, it can obtain write access to the primary buffer and mix streams directly into it. This feature is provided for completeness, and should only be useful for a very limited set of high-performance applications. Applications that take advantage of this feature are subject to very stringent performance requirements, since it is difficult to avoid gaps in the audio.

To implement a custom mixer, the application should first obtain the DSSCL_WRITEPRIMARY cooperative level and then create a primary sound buffer. It can then call the IDirectSoundBuffer::Lock method, write data into the returned pointers, and call the IDirectSoundBuffer::Unlock method to release the data back to DirectSound. Applications must explicitly play the primary buffer by calling the IDirectSoundBuffer::Play method to reproduce the sound data in the speakers. Note that the DSBPLAY_LOOPING flag must be specified or the **IDirectSoundBuffer::Play** call will fail.

The following example illustrates how an application might implement a custom mixer. This function would have to be called at regular intervals, frequently enough to prevent the sound device from repeating blocks of data. The **CustomMixer()** function is an application-defined function which mixes several streams together, as specified in the application-defined **AppStreamInfo** structure, and writes the result into the pointer specified:

```
BOOL AppMixIntoPrimaryBuffer(
    LPAPPSTREAMINFO lpAppStreamInfo,
    LPDIRECTSOUNDBUFFER lpDsbPrimary,
    DWORD dwDataBytes,
    DWORD dwOldPos,
    LPDWORD lpdwNewPos)
{
    LPVOID lpvPtr1;
    DWORD dwBytes1;
    LPVOID lpvPtr2;
    DWORD dwBytes2;
    HRESULT hr;
    // Obtain write pointer.
    hr = lpDsbPrimary->lpVtbl->Lock(lpDsbPrimary, dwOldPos, dwDataBytes,
        &lpvPtr1, &dwBytes1, &lpvPtr2, &dwBytes2, 0);
    // If we got DSERR_BUFFERLOST, restore and retry lock.
    if(DSERR_BUFFERLOST == hr) {
        lpDsbPrimary->lpVtbl->Restore(lpDsbPrimary);
        hr = lpDsbPrimary->lpVtbl->Lock(lpDsbPrimary, dwOldPos,
            dwDataBytes, &lpvPtr1, &dwBytes1, &lpvPtr2, &dwBytes2, 0);
    }
    if(DS_OK == hr) {
        // Mix data into the returned pointers.
        CustomMixer(lpAppStreamInfo, lpvPtr1, dwBytes1);
        *lpdwNewPos = dwOldPos + dwBytes1;
        if(NULL != lpvPtr2) {
            CustomMixer(lpAppStreamInfo, lpvPtr2, dwBytes2);
            *lpdwNewPos = dwBytes2; // Because we wrapped around.
        }
        // Release the data back to DirectSound.
        hr = lpDsbPrimary->lpVtbl->Unlock(lpDsbPrimary, lpvPtr1,
            dwBytes1, lpvPtr2, dwBytes2);
        if(DS_OK == hr) {
            // Success!
            return TRUE;
        }
    }
    // If we got here, then we failed Lock or Unlock.
    return FALSE;
```

```
}
```

Using Compressed Wave Formats

DirectSound does not currently support compressed wave formats. Applications should use the Audio Compression Manager (ACM) functions, provided with the Win32 SDK, to convert compressed audio to pulse coded modulation (PCM) data before writing the data to a sound buffer. In fact, by locking a pointer to the sound buffer memory and passing this pointer to the ACM, the data can be decoded directly into the sound buffer for maximum efficiency.

Functions

The DirectSound functions create a DirectSound object, which in turn controls the direct-audio memory access enabled by the DirectX 2 SDK. DirectSound virtualizes direct memory access (DMA) to the audio buffer of the sound card. This allows the application, using the DirectSound object, to mix and play its own audio.

DirectSoundCreate

```
HRESULT DirectSoundCreate(GUID FAR * lpGuid,
    LPDIRECTSOUND * ppDS, IUnknown FAR * pUnkOuter);
```

Creates and initializes an IDirectSound interface.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_ALLOCATED DSERR_INVALIDPARAM
  DSERR_NOAGGREGATION DSERR_NODRIVER
  DSERR_OUTOFMEMORY

lpGuid
Address of the GUID that identifies the sound device. The value of this parameter must be one of the GUIDs returned by DirectSoundEnumerate, or, to request the default device, NULL.

ppDS
Address of a pointer to a DirectSound object created in response to this function.

pUnkOuter
Controlling unknown of the aggregate. Its value must be NULL.

The application must call the IDirectSound::SetCooperativeLevel method immediately after creating a DirectSound object.

DirectSoundEnumerate

```
BOOL DirectSoundEnumerate(
    LPDSENUMCALLBACK lpDSEnumCallback, LPVOID lpContext);
```

Enumerates the DirectSound drivers installed in the system.

- Returns DS_OK if successful, or DSERR_INVALIDPARAM otherwise.

lpDSEnumCallback
Address of the DSEnumCallback function that will be called for each DirectSound object installed in the system.

lpContext
Address of the user-defined context passed to the enumeration callback function every time that function is called.

Callback Functions

Most of the functionality of DirectSound is provided by the methods of its Component Object Model (COM) interfaces. This section lists the callback functions that are not implemented as part of a COM interface.

DSEnumCallback

```
BOOL DSEnumCallback(GUID FAR * lpGuid,
    LPSTR lpstrDescription, LPSTR lpstrModule,
    LPVOID lpContext);
```

Application-defined callback function that enumerates the DirectSound drivers. The system calls this function in response to the application's previous call to the DirectSoundEnumerate function.

- Returns TRUE to continue enumerating drivers, or FALSE to stop.

lpGuid

Address of the GUID that identifies the DirectSound driver being enumerated. This value can be passed to the DirectSoundCreate function to create a DirectSound object for that driver.

lpstrDescription

Address of a null-terminated string that provides a textual description of the DirectSound device.

lpstrModule

Address of a null-terminated string that specifies the module name of the DirectSound driver corresponding to this device.

lpContext

Address of application-defined data that is passed to each callback function.

The application can save the strings passed in the *lpstrDescription* and *lpstrModule* parameters by copying them into memory that is allocated from the heap. The memory used to pass the strings to this callback function is valid only while this callback function is running.

IDirectSound Interface Method Groups

Applications use the methods of the IDirectSound interface to create DirectSound objects and set up the environment. The methods can be organized into the following groups:

Allocating memory	Compact Initialize
Creating buffers	CreateSoundBuffer DuplicateSoundBuffer SetCooperativeLevel
Device capabilities	GetCaps
IUnknown	AddRef QueryInterface Release
Speaker configuration	GetSpeakerConfig SetSpeakerConfig

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectSound object without affecting the functionality of the original interface.

IDirectSound::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectSound object by 1. This method is part of the IUnknown interface inherited by DirectSound.

- Returns the new reference count of the object.

When the DirectSound object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirectSound::Release method to decrease the reference count of the object by 1.

IDirectSound::Compact

```
HRESULT Compact();
```

Moves the unused portions of onboard sound memory, if any, to a contiguous block so that the largest portion of free memory will be available.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED
  DSERR_UNINITIALIZED

If the application calls this method, it must have exclusive cooperation with the DirectSound object. (To get exclusive access, specify DSSCL_EXCLUSIVE in a call to the IDirectSound::SetCooperativeLevel method.) This method will fail if any operations are in progress.

IDirectSound::CreateSoundBuffer

```
HRESULT CreateSoundBuffer(LPDSBUFFERDESC lpDSBufferDesc,
    LPLPDIRECTSOUNDBUFFER * lplpDirectSoundBuffer,
    IUnknown FAR * pUnkOuter);
```

Creates a DirectSoundBuffer object to hold a sequence of audio samples.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_ALLOCATED DSERR_BADFORMAT
  DSERR_INVALIDPARAM DSERR_NOAGGREGATION
  DSERR_OUTOFMEMORY DSERR_UNINITIALIZED
  DSERR_UNSUPPORTED

lpDSBufferDesc
Address of a DSBUFFERDESC structure that contains the description of the sound buffer to be created.

lplpDirectSoundBuffer
Address of a pointer to the new DirectSoundBuffer object or NULL if the buffer cannot be created.

pUnkOuter
Controlling unknown of the aggregate. Its value must be NULL.

Before it can play any sound buffers, the application must specify a cooperative level for a DirectSound object by using the IDirectSound::SetCooperativeLevel method.

The *lpDSBufferDesc* parameter points to a structure that describes the type of buffer desired, including format, size, and capabilities. The application must specify the needed capabilities, or they will not be available. For example, if the application creates a DirectSoundBuffer object without specifying the DSBCAPS_CTRLFREQUENCY flag, any call to IDirectSoundBuffer::SetFrequency will fail.

The DSBCAPS_STATIC flag can also be specified, in which case DirectSound stores the buffer in onboard memory, if available, in order to take advantage of hardware mixing. To force the buffer to use either hardware or software mixing, use the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flags.

IDirectSound::DuplicateSoundBuffer

```
HRESULT DuplicateSoundBuffer(
    LPDIRECTSOUNDBUFFER lpDsbOriginal,
    LPLPDIRECTSOUNDBUFFER lplpDsbDuplicate);
```

Creates a new DirectSoundBuffer object that uses the same buffer memory as the original object.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_ALLOCATED DSERR_INVALIDCALL
  DSERR_INVALIDPARAM DSERR_OUTOFMEMORY
  DSERR_UNINITIALIZED

lpDsbOriginal
Address of the DirectSoundBuffer object to be duplicated.

lplpDsbDuplicate
Address of a pointer to the new DirectSoundBuffer object.

The new object can be used just like the original.

Initially, the duplicate buffer will have the same parameters as the original buffer. However, the application can change the parameters of each buffer independently, and each can be played or stopped without affecting the other.

If data in the buffer is changed through one object, the change will be reflected in the other object since the buffer memory is shared.

The buffer memory will be released when the last object referencing it is released.

IDirectSound::GetCaps

```
HRESULT GetCaps(LPDSCAPS lpDSCaps);
```

Retrieves the capabilities of the hardware device that is represented by the DirectSound object.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_GENERIC DSERR_INVALIDPARAM
  DSERR_UNINITIALIZED

lpDSCaps
Address of the DSCAPS structure to contain the capabilities of this sound device.

Information retrieved in the **DSCAPS** structure describes the maximum capabilities of the sound device and those currently available, such as the number of hardware mixing channels and the amount of onboard sound memory. This information can be used to fine-tune performance and optimize resource allocation.

Because of resource sharing requirements, the maximum capabilities in one area might only be available at the cost of another area. For example, the maximum number of hardware-mixed streaming buffers may only be available if there are no hardware static buffers.

IDirectSound::GetSpeakerConfig

```
HRESULT GetSpeakerConfig(LPDWORD lpdwSpeakerConfig);
```

Retrieves the speaker configuration specified for this DirectSound object.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDPARAM DSERR_UNINITIALIZED

lpdwSpeakerConfig

Address of the speaker configuration for this DirectSound object. The speaker configuration is specified with one of the following values:

DSSPEAKER_HEADPHONE

The audio is output through headphones.

DSSPEAKER_MONO

The audio is output through a single speaker.

DSSPEAKER_QUAD

The audio is output through quadraphonic speakers.

DSSPEAKER_STEREO

The audio is output through stereo speakers (default value).

DSSPEAKER_SURROUND

The audio is output through surround speakers.

IDirectSound::Initialize

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

Initializes the DirectSound object if it has not yet been initialized.

- Returns DSERR_ALREADYINITIALIZED.

lpGuid

Address of the GUID specifying the sound driver for this DirectSound object to bind to, or NULL to select the primary sound driver.

Because the DirectSoundCreate function calls this method internally, it is not needed for the current release of DirectSound.

IDirectSound::QueryInterface

```
HRESULT QueryInterface(REFIID  riid, LPVOID * ppvObj);
```

Determines if the DirectSound object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by DirectSound.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_GENERIC DSERR_INVALIDPARAM

riid
Reference identifier of the interface being requested.

ppvObj
Address of a location to be filled with the returned interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectSound::QueryInterface** method allows DirectSound objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirectSound::Release

```
ULONG Release();
```

Decreases the reference count of the DirectSound object by 1. This method is part of the IUnknown interface inherited by DirectSound.

- Returns the new reference count of the object.

The DirectSound object deallocates itself when its reference count reaches 0. Use the IDirectSound::AddRef method to increase the reference count of the object by 1.

IDirectSound::SetCooperativeLevel

```
HRESULT SetCooperativeLevel(HWND hwnd, DWORD dwLevel);
```

Sets the cooperative level of the application for this sound device.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_ALLOCATED DSERR_INVALIDPARAM
  DSERR_UNINITIALIZED DSERR_UNSUPPORTED

hwnd

Window handle for the application.

dwLevel

Requested priority level. Specify one of the following values:

DSSCL_EXCLUSIVE

Sets the application to the exclusive level. When it has the input focus, the application will be the only one audible. With this level, it also has all the privileges of the DSSCL_PRIORITY level. DirectSound will restore the hardware format, as specified by the most recent call to the IDirectSoundBuffer::SetFormat method, once the application gains the input focus.

DSSCL_NORMAL

Sets the application to a fully cooperative status. Most applications should use this level, since it has the smoothest multi-tasking and resource-sharing behavior.

DSSCL_PRIORITY

Sets the application to the priority level. Applications with this cooperative level can call the **IDirectSoundBuffer::SetFormat** and IDirectSound::Compact methods.

DSSCL_WRITEPRIMARY

This is the highest priority level. The application has write access to the primary buffers. No secondary buffers in any application can be played.

The application must set the cooperative level by calling this method before its buffers can be played. The recommended cooperative level is DSSCL_NORMAL; use other priority levels when necessary.

Four cooperative levels are defined: normal, priority, exclusive, and write-primary.

The normal cooperative level is the lowest level. At the normal level, the **IDirectSoundBuffer::SetFormat** and **IDirectSound::Compact** methods cannot be called. In addition, the application cannot obtain write access to primary buffers. All applications using this cooperative level use a primary buffer format of 22 kHz, monaural sound, and 8-bit samples to make task switching as smooth as possible.

When using a DirectSound object with the priority cooperative level, the application has first priority to hardware resources, such as hardware mixing, and can call **IDirectSoundBuffer::SetFormat** and IDirectSound::Compact.

When using a DirectSound object with the exclusive cooperative level, the application has all the privileges of the priority level; in addition, when it has the input focus, only this application's buffers are audible. Once the input focus is gained, DirectSound restores the application's preferred wave format, which was defined in the most recent call to IDirectSoundBuffer::SetFormat.

The highest cooperative level is write-primary. When using a DirectSound object with this cooperative level, your application has direct access to the primary sound buffer. In this mode, the application must lock the buffer using the IDirectSoundBuffer::Lock method and write directly into the primary buffer. While this occurs, secondary buffers cannot be played.

When the application is set to the write-primary cooperative level and gains the input focus, all secondary buffers for other applications are stopped and marked as lost. These buffers must be restored using the IDirectSoundBuffer::Restore method before they can be played again. When the application then loses the input focus, its primary buffer is marked as lost and can be restored after the application regains the input focus.

The write-primary level is not required in order to create a primary buffer. However, to obtain access to the audio samples in the primary buffer, the application must have the write-primary level. If the application does not have this level, then all calls to **IDirectSoundBuffer::Lock** and IDirectSoundBuffer::Play will fail, although some methods, such as IDirectSoundBuffer::GetFormat, **IDirectSoundBuffer::SetFormat**, and IDirectSoundBuffer::GetVolume, can still be called successfully.

IDirectSound::SetSpeakerConfig

```
HRESULT SetSpeakerConfig(DWORD dwSpeakerConfig);
```

Specifies the speaker configuration for the DirectSound object.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDPARAM DSERR_UNINITIALIZED

dwSpeakerConfig

Speaker configuration of the specified DirectSound object. This parameter can be one of the following values:

DSSPEAKER_HEADPHONE

The speakers are headphones.

DSSPEAKER_MONO

The speakers are monaural.

DSSPEAKER_QUAD

The speakers are quadraphonic.

DSSPEAKER_STEREO

The speakers are stereo (default value).

DSSPEAKER_SURROUND

The speakers are surround sound.

IDirectSoundBuffer Interface Method Groups

Applications use the methods of the **IDirectSoundBuffer** interface to create DirectSoundBuffer objects and set up the environment. The methods can be organized into the following groups:

Information	GetCaps GetFormat GetStatus SetFormat
IUnknown	AddRef QueryInterface Release
Memory management	Initialize Restore
Play management	GetCurrentPosition Lock Play SetCurrentPosition Stop Unlock
Sound management	GetFrequency GetPan GetVolume SetFrequency SetPan SetVolume

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectSoundBuffer object without affecting the functionality of the original interface.

IDirectSoundBuffer::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectSoundBuffer object by 1. This method is part of the IUnknown interface inherited by DirectSound.

- Returns the new reference count of the object.

When the DirectSoundBuffer object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirectSoundBuffer::Release method to decrease the reference count of the object by 1.

IDirectSoundBuffer::GetCaps

```
HRESULT GetCaps(LPDSBCAPS lpDSBufferCaps);
```

Retrieves the capabilities of the DirectSoundBuffer object.

- Returns DS_OK if successful, or DSERR_INVALIDPARAM otherwise.

lpDSBufferCaps

Address of a DSBCAPS structure to contain the capabilities of this sound buffer.

The **DSBCAPS** structure contains similar information to the DSBUFFERDESC structure passed to the IDirectSound::CreateSoundBuffer method, with some additional information. This additional information can include the location of the buffer, either in hardware or software, and some cost measures. Examples of cost measures include the time it takes to download to a hardware buffer and the processing overhead required to mix and play the buffer when it is in the system memory.

The flags specified in the **dwFlags** member of the **DSBCAPS** structure are the same flags used by the **DSBUFFERDESC** structure. The only difference is that in the **DSBCAPS** structure, either DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE will be specified according to the location of the buffer memory. In the **DSBUFFERDESC** structure, these flags are optional and, depending on which flag is specified, force the buffer to be located in either hardware or software.

IDirectSoundBuffer::GetCurrentPosition

```
HRESULT GetCurrentPosition(LPDWORD lpdwCurrentPlayCursor,
    LPDWORD lpdwCurrentWriteCursor);
```

Retrieves the current position of the play and write cursors in the sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

lpdwCurrentPlayCursor
Address of a variable to contain the current play position in the DirectSoundBuffer object. This position is an offset within the sound buffer and is specified in bytes.

lpdwCurrentWriteCursor
Address of a variable to contain the current write position in the DirectSoundBuffer object. This position is an offset within the sound buffer and is specified in bytes.

The write cursor indicates the position at which it is safe to write new data into the buffer. The write cursor always leads the play cursor, typically by about 15 milliseconds worth of audio data.

It is always safe to change data that is behind the position indicated by the *lpdwCurrentPlayCursor* parameter.

IDirectSoundBuffer::GetFormat

```
HRESULT GetFormat(LPWAVEFORMATEX lpwfxFormat,
    DWORD dwSizeAllocated, LPDWORD lpdwSizeWritten);
```

Retrieves a description of the format of the sound data in the buffer, or the buffer size needed to retrieve the format description.

- Returns DS_OK if successful, or DSERR_INVALIDPARAM otherwise.

lpwfxFormat

Address of the WAVEFORMATEX structure to contain a description of the sound data in the buffer. To retrieve the buffer size needed to contain the format description, specify NULL.

dwSizeAllocated

Size, in bytes, of the WAVEFORMATEX structure. DirectSound writes, at most, *dwSizeAllocated* bytes to that pointer; if the WAVEFORMATEX structure requires more memory, it is truncated.

lpdwSizeWritten

Address of a variable to contain the number of bytes written to the WAVEFORMATEX structure. This parameter can be NULL.

The WAVEFORMATEX structure can have a variable length that depends on the details of the format. Before retrieving the format description, the application should query the DirectSoundBuffer object for the size of the format by calling this method and specifying NULL for the *lpwfxFormat* parameter. The size of the structure will be returned in the *lpdwSizeWritten* parameter. The application can then allocate sufficient memory and call **IDirectSoundBuffer::GetFormat** again to retrieve the format description.

IDirectSoundBuffer::GetFrequency

```
HRESULT GetFrequency(LPDWORD lpdwFrequency);
```

Retrieves the frequency, in samples per second, at which the buffer is being played. This value will be in the range of 100-100,000.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_CONTROLUNAVAIL DSERR_INVALIDPARAM
  DSERR_PRIOLEVELNEEDED

lpdwFrequency
Address of the variable that represents the frequency at which the audio buffer is being played.

IDirectSoundBuffer::GetPan

```
HRESULT GetPan(LPLONG lplPan);
```

Retrieves a variable that represents the relative volume between the left and right audio channels.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_CONTROLUNAVAIL DSERR_INVALIDPARAM
  DSERR_PRIOLEVELNEEDED

lplPan

Address of a variable to contain the relative mix between the left and right speakers.

The returned value is measured in hundredths of a decibel (dB), in the range of -10,000 to 10,000. The value -10,000 means the right channel is attenuated by 100 dB. The value 10,000 means the left channel is attenuated by 100 dB. 0 is the neutral value; a pan value of 0 means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than 0, one of the channels is at full volume and the other is attenuated.

A pan of -2173 means that the left channel is at full volume and the right channel is attenuated by 21.73 dB. Similarly, a pan of 870 means that the left channel is attenuated by 8.7 dB and the right channel is at full volume. A pan of -10,000 means that the right channel is silent and the sound is "all the way to the left," while a pan of 10,000 means that the left channel is silent and the sound is "all the way to the right."

The pan control acts cumulatively with the volume control.

IDirectSoundBuffer::GetStatus

```
HRESULT GetStatus(LPDWORD lpdwStatus);
```

Retrieves the current status of the sound buffer.

- Returns DS_OK if successful, or DSERR_INVALIDPARAM otherwise.

lpdwStatus

Address of a variable to contain the status of the sound buffer. The status can be set to one of the following values:

DSBSTATUS_BUFFERLOST

The buffer is lost and must be restored before it can be played or locked.

DSBSTATUS_LOOPING

The buffer is being looped. If this value is not set, the buffer will stop when it reaches the end of the sound data. Note that if this value is set, the buffer must also be playing.

DSBSTATUS_PLAYING

The buffer is being played. If this value is not set, the buffer is stopped.

IDirectSoundBuffer::GetVolume

```
HRESULT GetVolume(LPLONG lplVolume);
```

Retrieves the current volume for this sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_CONTROLUNAVAIL DSERR_INVALIDPARAM
  DSERR_PRIOLEVELNEEDED

lplVolume

Address of the variable to contain the volume associated with the specified DirectSound buffer.

The volume is specified in hundredths of decibels (dB), and ranges from 0 to -10,000. The value 0 represents the original, unadjusted volume of the stream. The value -10,000 indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Amplification is not currently supported by DirectSound.

The decibel (dB) scale corresponds to the logarithmic hearing characteristics of the ear. For example, an attenuation of 10 dB makes a buffer sound half as loud, and an attenuation of 20 dB makes a buffer sound one quarter as loud.

IDirectSoundBuffer::Initialize

```
HRESULT Initialize(LPDIRECTSOUND lpDirectSound,
    LPDSBUFFERDESC lpDSBufferDesc);
```

Initializes a DirectSoundBuffer object if it has not yet been initialized.

- Returns DSERR_ALREADYINITIALIZED.

lpDirectSound

Address of the DirectSound object associated with this DirectSoundBuffer object.

lpDSBufferDesc

Address of a DSBUFFERDESC structure that contains the values used to initialize this sound buffer.

Because the IDirectSound::CreateSoundBuffer method calls **IDirectSoundBuffer::Initialize** internally, it is not needed for the current release of DirectSound. This method is provided for future extensibility.

IDirectSoundBuffer::Lock

```
HRESULT Lock(DWORD dwWriteCursor, DWORD dwWriteBytes,
    LPVOID lplpvAudioPtr1, LPDWORD lpdwAudioBytes1,
    LPVOID lplpvAudioPtr2, LPDWORD lpdwAudioBytes2,
    DWORD dwFlags);
```

Obtains a valid write pointer to the sound buffer's audio data.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_BUFFERLOST DSERR_INVALIDCALL
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

dwWriteCursor
Offset, in bytes, from the start of the buffer to where the lock begins. This parameter is ignored if DSBLOCK_FROMWRITECURSOR is specified in the *dwFlags* parameter.

dwWriteBytes
Size, in bytes, of the portion of the buffer to lock. Note that the sound buffer is conceptually circular.

lplpvAudioPtr1
Address of a pointer to contain the first block of the sound buffer to be locked.

lpdwAudioBytes1
Address of a variable to contain the number of bytes pointed to by the *lplpvAudioPtr1* parameter. If this value is less than the *dwWriteBytes* parameter, *lplpvAudioPtr2* will point to a second block of sound data.

lplpvAudioPtr2
Address of a pointer to contain the second block of the sound buffer to be locked. If the value of this parameter is NULL, the *lplpvAudioPtr1* parameter points to the entire locked portion of the sound buffer.

lpdwAudioBytes2
Address of a variable to contain the number of bytes pointed to by the *lplpvAudioPtr2* parameter. If *lplpvAudioPtr2* is NULL, this value will be 0.

dwFlags
Flags modifying the lock event. The following flag is defined:

DSBLOCK_FROMWRITECURSOR
Locks from the current write cursor, making a call to IDirectSoundBuffer::GetCurrentPosition unnecessary. If this flag is specified, the *dwWriteCursor* parameter is ignored. This flag is optional.

This method accepts an offset and a byte count, and returns two write pointers and their associated sizes. Two pointers are required because sound buffers are circular. If the locked bytes do not wrap around the end of the buffer, the second pointer, *lplpvAudioBytes2*, will be NULL. However, if the bytes do wrap around, then the second pointer will point to the beginning of the buffer.

If the application passes NULL for the *lplpvAudioPtr2* and *lpdwAudioBytes2* parameters, DirectSound will not lock the wraparound portion of the buffer.

The application should write data into the pointers returned by the **IDirectSoundBuffer::Lock** method and then call the IDirectSoundBuffer::Unlock method to release the buffer back to DirectSound. The sound buffer should not be locked for long periods of time; if it is, the play cursor will reach the locked bytes and configuration-dependent audio problems, possibly random noise, will result.

Warning This method returns a write pointer only. The application should not try to read sound data from this pointer; the data may not be valid even though the DirectSoundBuffer object contains valid sound data. For example, if the buffer is located in onboard memory, the pointer may be an address to a temporary buffer in main system memory. When **IDirectSoundBuffer::Unlock** is called, this temporary buffer will be transferred to the onboard memory.

IDirectSoundBuffer::Play

```
HRESULT Play(DWORD dwReserved1, DWORD dwReserved2,
    DWORD dwFlags);
```

Causes the sound buffer to play from the current position.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_BUFFERLOST DSERR_INVALIDCALL
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

dwReserved1

This parameter is reserved. Its value must be 0.

dwReserved2

This parameter is reserved. Its value must be 0.

dwFlags

Flags specifying how to play the buffer. The following flag is defined:

DSBPLAY_LOOPING

Once the end of the audio buffer is reached, play restarts at the beginning of the buffer. Play continues until explicitly stopped. This flag must be set when playing primary buffers.

For secondary buffers, this method will cause the buffer to be mixed into the primary buffer and output to the sound device. If this is the first buffer to play, it will implicitly create a primary buffer and start playing that buffer; the application does not need to explicitly direct the primary buffer to play.

If the buffer specified in the method is already playing, the call to the method will succeed and the buffer will continue to play. However, the flags that define playback characteristics are superseded by the flags defined in the most recent call.

Primary buffers must be played with the DSBPLAY_LOOPING flag set.

For primary sound buffers, this method will cause them to start playing to the sound device. If the application is set to the DSSCL_WRITEPRIMARY cooperative level, this will cause the audio data in the primary buffer to be output to the sound device. However, if the application is set to any other cooperative level, this method will ensure that the primary buffer is playing even when no secondary buffers are playing; in that case, silence will be played. This may reduce processing overhead when sounds are started and stopped in sequence since the primary buffer will be playing continuously rather than stopping and starting between secondary buffers.

Note Before this method can be called on any sound buffer, the application must call the IDirectSound::SetCooperativeLevel method and specify a cooperative level, typically DSSCL_NORMAL. If **IDirectSound::SetCooperativeLevel** has not been called, the **IDirectSoundBuffer::Play** method returns the DSERR_PRIOLEVELNEEDED error value.

IDirectSoundBuffer::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID * ppvObj);
```

Determines if the DirectSoundBuffer object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by DirectSound.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDPARAM DSERR_GENERIC

riid

Reference identifier of the interface being requested.

ppvObj

Address of a location that will be filled with the returned interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectSoundBuffer::QueryInterface** method allows DirectSoundBuffer objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirectSoundBuffer::Release

```
ULONG Release();
```

Decreases the reference count of the DirectSoundBuffer object by 1. This method is part of the IUnknown interface inherited by DirectSound.

- Returns the new reference count of the object.

The DirectSound object deallocates itself when its reference count reaches 0. Use the IDirectSoundBuffer::AddRef method to increase the reference count of the object by 1.

IDirectSoundBuffer::Restore

```
HRESULT Restore();
```

Restores the memory allocation for a lost sound buffer for the specified DirectSoundBuffer object.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_BUFFERLOST DSERR_INVALIDCALL
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

If the application does not have the input focus, **IDirectSoundBuffer::Restore** may not succeed. For example, if the application with the input focus has the DSSCL_WRITEPRIMARY cooperative level, no other application will be able to restore its buffers. Similarly, an application with the DSSCL_WRITEPRIMARY cooperative level must have the input focus to restore its primary buffer.

Once DirectSound restores the buffer memory, the application must rewrite the buffer with valid sound data. DirectSound cannot restore the contents of the memory, only the memory itself.

The application can receive notification that a buffer is lost when it specifies that buffer in a call to the IDirectSoundBuffer::Lock or IDirectSoundBuffer::Play method. These methods return DSERR_BUFFERLOST to indicate a lost buffer. The IDirectSoundBuffer::GetStatus method can also be used to retrieve the status of the sound buffer and test for the DSBSTATUS_BUFFERLOST flag.

IDirectSoundBuffer::SetCurrentPosition

```
HRESULT SetCurrentPosition(DWORD dwNewPosition);
```

Moves the current play cursor for secondary sound buffers.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDCALL DSERR_INVALIDPARAM
  DSERR_PRIOLEVELNEEDED

dwNewPosition

New position, in bytes, from the beginning of the buffer that will be used when the sound buffer is played.

This method cannot be called on primary sound buffers.

If the buffer is playing, it will immediately move to the new position and continue. If it is not playing, it will begin from the new position the next time the IDirectSoundBuffer::Play method is called.

IDirectSoundBuffer::SetFormat

```
HRESULT SetFormat(LPWAVEFORMATEX lpfxFormat);
```

Sets the format of the primary sound buffer for the application. Whenever this application has the input focus, DirectSound will set the primary buffer to the specified format.

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

DSERR_BADFORMAT	DSERR_INVALIDCALL
DSERR_INVALIDPARAM	DSERR_OUTOFMEMORY
DSERR_PRIOLEVELNEEDED	DSERR_UNSUPPORTED

lpfxFormat

Address of a WAVEFORMATEX structure that describes the new format for the primary sound buffer.

A call to this method fails if the sound buffer is playing or the hardware does not directly support the requested pulse coded modulation (PCM) format. It will also fail if the calling application has the DSSCL_NORMAL cooperative level.

If a secondary buffer requires a format change, the application should create a new DirectSoundBuffer object using the new format.

DirectSound supports PCM formats; it does not currently support compressed formats.

IDirectSoundBuffer::SetFrequency

```
HRESULT SetFrequency(DWORD dwFrequency);
```

Sets the frequency at which the audio samples are played.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_CONTROLUNAVAIL DSERR_GENERIC
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

dwFrequency

New frequency, in Hz, at which to play the audio samples. The value must be between 100 and 100,000.

If the value is 0, the frequency is reset to the current buffer format. This format is specified in the IDirectSound::CreateSoundBuffer method.

Increasing or decreasing the frequency changes the perceived pitch of the audio data. This method does not affect the format of the buffer.

IDirectSoundBuffer::SetPan

```
HRESULT SetPan(LONG lPan);
```

Specifies the relative volume between the left and right channels.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_CONTROLUNAVAIL DSERR_GENERIC
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

lPan

Relative volume between the left and right channels. This value has a range of -10,000 to 10,000 and is measured in hundredths of a decibel.

0 is the neutral value for *lPan* and indicates that both channels are at full volume (attenuated by 0 decibels). At any other setting, one of the channels is at full volume and the other is attenuated. For example, a pan of -2173 means that the left channel is at full volume and the right channel is attenuated by 21.73 dB. Similarly, a pan of 870 means that the left channel is attenuated by 8.7 dB and the right channel is at full volume.

A pan of -10,000 means that the right channel is silent and the sound is "all the way to the left," while a pan of 10,000 means that the left channel is silent and the sound is "all the way to the right."

The pan control is cumulative with the volume control.

IDirectSoundBuffer::SetVolume

```
HRESULT SetVolume(LONG lVolume);
```

Changes the volume of a sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_CONTROLUNAVAIL DSERR_GENERIC
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

lVolume

New volume requested for this sound buffer. Values range from 0 (0 dB, no volume adjustment) to -10,000 (-100 dB, essentially silent). DirectSound does not currently support amplification.

Volume units of are in hundredths of decibels, where 0 is the original volume of the stream.

Positive decibels correspond to amplification and negative decibels correspond to attenuation. The decibel scale corresponds to the logarithmic hearing characteristics of the ear. An attenuation of 10 dB makes a buffer sound half as loud; an attenuation of 20 dB makes a buffer sound one quarter as loud. DirectSound does not currently support amplification.

The pan control is cumulative with the volume control.

IDirectSoundBuffer::Stop

```
HRESULT Stop();
```

Causes the sound buffer to stop playing.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDPARAM DSERR_PRIOLEVELNEEDED

For secondary buffers, **IDirectSoundBuffer::Stop** will set the current position of the buffer to the sample that follows the last sample played. This means that if the IDirectSoundBuffer::Play method is called on the buffer, it will continue playing where it left off.

For primary buffers, if an application has the DSSCL_WRITEPRIMARY level, this method will stop the buffer and reset the current position to 0 (the beginning of the buffer). This is necessary because the primary buffers on most sound cards can only play from the beginning of the buffer.

However, if **IDirectSoundBuffer::Stop** is called on a primary buffer and the application has a cooperative level other than DSSCL_WRITEPRIMARY, this method simply reverses the effects of **IDirectSoundBuffer::Play**. It configures the primary buffer to stop if no secondary buffers are playing. If other buffers are playing in this or other applications, the primary buffer will not actually stop until they are stopped. This method is useful because playing the primary buffer consumes processing overhead even if the buffer is playing sound data with the amplitude of 0 dB.

IDirectSoundBuffer::Unlock

```
HRESULT Unlock(LPVOID lpvAudioPtr1, DWORD dwAudioBytes1,
    LPVOID lpvAudioPtr2, DWORD dwAudioBytes2);
```

Releases a locked sound buffer.

- Returns DS_OK if successful, or one of the following error values otherwise:
  DSERR_INVALIDCALL DSERR_INVALIDPARAM
  DSERR_PRIOLEVELNEEDED

lpvAudioPtr1
Address of the value retrieved in the *lplpvAudioPtr1* parameter of the IDirectSoundBuffer::Lock method.

dwAudioBytes1
Number of bytes actually written into the *lpvAudioPtr1* parameter. It should not exceed the number of bytes returned by the **IDirectSoundBuffer::Lock** method.

lpvAudioPtr2
Address of the value retrieved in the *lplpvAudioPtr2* parameter of the **IDirectSoundBuffer::Lock** method.

dwAudioBytes2
Number of bytes actually written into the *lpvAudioPtr2* parameter. It should not exceed the number of bytes returned by the **IDirectSoundBuffer::Lock** method.

An application must pass both pointers, *lpvAudioPtr1* and *lpvAudioPtr2*, returned by the **IDirectSoundBuffer::Lock** method to ensure the correct pairing of **IDirectSoundBuffer::Lock** and **IDirectSoundBuffer::Unlock**. The second pointer is needed even if 0 bytes were written to the second pointer.

Applications must pass the number of bytes actually written to the two pointers in the parameters *dwAudioBytes1* and *dwAudioBytes2*.

Make sure the sound buffer does not remain locked for long periods of time.

DSBCAPS

```
typedef struct _DSBCAPS {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwBufferBytes;
    DWORD dwUnlockTransferRate;
    DWORD dwPlayCpuOverhead;
} DSBCAPS, *LPDSBCAPS;
```

Specifies the capabilities of a DirectSound buffer object, for use by the IDirectSoundBuffer::GetCaps method.

dwSize

Size of this structure, in bytes.

dwFlags

Flags that specify buffer-object capabilities.

DSBCAPS_CTRLFREQUENCY

The buffer must have frequency control capability.

DSBCAPS_CTRLPAN

The buffer must have pan control capability.

DSBCAPS_CTRLVOLUME

The buffer must have volume control capability.

DSBCAPS_LOCHARDWARE

Forces the buffer to use hardware mixing, even if DSBCAPS_STATIC is not specified. If the device does not support hardware mixing, or the required hardware memory is not available, the call to IDirectSound::CreateSoundBuffer will fail. The application must ensure that a mixing channel will be available for this buffer; this condition is not guaranteed.

DSBCAPS_LOCSOFTWARE

Forces the buffer to be stored in software memory and use software mixing, even if DSBCAPS_STATIC is specified and hardware resources are available.

DSBCAPS_PRIMARYBUFFER

Indicates that the buffer is a primary sound buffer. If this value is not specified, a secondary buffer will be created.

DSBCAPS_STATIC

Indicates that the buffer will be used for static sound data. Typically, these buffers are loaded once and played many times. These buffers are candidates for hardware memory.

DSBCAPS_STICKYFOCUS

Changes the focus behavior of the sound buffer. This flag can be specified in an IDirectSound::CreateSoundBuffer call. With this flag set, an application using DirectSound can continue to play its sticky focus buffers if the user switches to another application not using DirectSound. In this situation, the application's normal buffers will be muted, but the sticky focus buffers will still be audible. This is useful for non-game applications, such as movie playback (ActiveMovie™) when the user wants to hear the soundtrack while typing in Word or Excel, for example. If the user switches to DirectSound, all sound buffers, both normal and sticky focus, in the previous application will be muted.

dwBufferBytes

Size of this buffer, in bytes.

dwUnlockTransferRate

Specifies the rate, in kilobytes per second, that data is transferred to the buffer memory when IDirectSoundBuffer::Unlock is called. High-performance applications can use this value to determine the time required for **IDirectSoundBuffer::Unlock** to execute. For software buffers located in system memory, the rate will be very high since no processing is required. For hardware buffers, the rate may be slower because the buffer might have to be downloaded to the sound card, which may have a limited transfer rate.

dwPlayCpuOverhead

Specifies the processing overhead as a percentage of main processing cycles needed to mix this sound buffer. For hardware buffers, this member will be 0 because the mixing is performed by the sound device. For software buffers, this member depends on the buffer format and the speed of the system processor.

The **DSBCAPS** structure contains information similar to that found in the DSBUFFERDESC structure passed to the IDirectSound::CreateSoundBuffer method, with some additional information. Additional information includes the location of the buffer (hardware or software) and some cost measures (such as the time to download the buffer if located in hardware, and the processing overhead to play the buffer if it is mixed in software).

Note that the **dwFlags** member of the **DSBCAPS** structure contains the same flags used by the **DSBUFFERDESC** structure. The only difference is that in the **DSBCAPS** structure, either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag will be specified, according to the location of the buffer memory. In the **DSBUFFERDESC** structure, these flags are optional and are used to force the buffer to be located in either hardware or software.

DSBUFFERDESC

```
typedef struct _DSBUFFERDESC{
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwBufferBytes;
    DWORD           dwReserved;
    LPWAVEFORMATEX  lpwfxFormat;
} DSBUFFERDESC, *LPDSBUFFERDESC;
```

Describes the necessary characteristics of a new DirectSoundBuffer object. This structure is used by the IDirectSound::CreateSoundBuffer method.

dwSize

Size of this structure, in bytes.

dwFlags

Identifies the capabilities to include when creating a new DirectSoundBuffer object. Specify one or more of the following:

DSBCAPS_CTRLALL

The buffer must have all control capabilities.

DSBCAPS_CTRLDEFAULT

The buffer should have default control options. This is the same as specifying the DSBCAPS_CTRLPAN, DSBCAPS_CTRLVOLUME, and DSBCAPS_CTRLFREQUENCY flags.

DSBCAPS_CTRLFREQUENCY

The buffer must have frequency control capability.

DSBCAPS_CTRLPAN

The buffer must have pan control capability.

DSBCAPS_CTRLVOLUME

The buffer must have volume control capability.

DSBCAPS_LOCHARDWARE

Forces the buffer to use hardware mixing, even if DSBCAPS_STATIC is not specified. If the device does not support hardware mixing or if the required hardware memory is not available, the call to the IDirectSound::CreateSoundBuffer method will fail. The application must ensure that a mixing channel will be available for this buffer; this condition is not guaranteed.

DSBCAPS_LOCSOFTWARE

Forces the buffer to be stored in software memory and use software mixing, even if DSBCAPS_STATIC is specified and hardware resources are available.

DSBCAPS_PRIMARYBUFFER

Indicates that the buffer is a primary sound buffer. If this value is not specified, a secondary buffer will be created.

DSBCAPS_STATIC

Indicates that the buffer will be used for static sound data. Typically, these buffers are loaded once and played many times. These buffers are candidates for hardware memory.

dwBufferBytes

Size of the new buffer, in bytes. This value must be 0 when creating primary buffers.

dwReserved

This value is reserved. Do not use.

lpwfxFormat

Address of a structure specifying the waveform format for the buffer. This value must be NULL for primary buffers. The application can use IDirectSoundBuffer::SetFormat to set the format of the primary buffer.

The DSBCAPS_LOCHARDWARE and DSBCAPS_LOCSOFTWARE flags used in the **dwFlags**

member are optional and mutually exclusive. DSBCAPS_LOCHARDWARE forces the buffer to reside in memory located in the sound card. DSBCAPS_LOCSOFTWARE forces the buffer to reside in main system memory, if possible.

These flags are also defined for the **dwFlags** member of the DSBCAPS structure, and when used there, the specified flag indicates the actual location of the DirectSoundBuffer object.

When creating a primary buffer, applications must set the **dwBufferBytes** member to 0; DirectSound will determine the optimal buffer size for the particular sound device in use. To determine the size of a created primary buffer, call IDirectSoundBuffer::GetCaps.

DSCAPS

```
typedef struct _DSCAPS {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwMinSecondarySampleRate;
    DWORD  dwMaxSecondarySampleRate;
    DWORD  dwPrimaryBuffers;
    DWORD  dwMaxHwMixingAllBuffers;
    DWORD  dwMaxHwMixingStaticBuffers;
    DWORD  dwMaxHwMixingStreamingBuffers;
    DWORD  dwFreeHwMixingAllBuffers;
    DWORD  dwFreeHwMixingStaticBuffers;
    DWORD  dwFreeHwMixingStreamingBuffers;
    DWORD  dwMaxHw3DAllBuffers;
    DWORD  dwMaxHw3DStaticBuffers;
    DWORD  dwMaxHw3DStreamingBuffers;
    DWORD  dwFreeHw3DAllBuffers;
    DWORD  dwFreeHw3DStaticBuffers;
    DWORD  dwFreeHw3DStreamingBuffers;
    DWORD  dwTotalHwMemBytes;
    DWORD  dwFreeHwMemBytes;
    DWORD  dwMaxContigFreeHwMemBytes;
    DWORD  dwUnlockTransferRateHwBuffers;
    DWORD  dwPlayCpuOverheadSwBuffers;
    DWORD  dwReserved1;
    DWORD  dwReserved2;
} DSCAPS, *LPDSCAPS;
```

Specifies the capabilities of a DirectSound device for use by the IDirectSound::GetCaps method.

dwSize

Size of this structure, in bytes.

dwFlags

Specifies device capabilities. Can be one or more of the following:

DSCAPS_CERTIFIED

This driver has been tested and certified by Microsoft.

DSCAPS_CONTINUOUSRATE

The device supports all sample rates between the **dwMinSecondarySampleRate** and **dwMaxSecondarySampleRate** member values. Typically, this means that the actual output rate will be within +/- 10 Hz of the requested frequency.

DSCAPS_EMULDRIVER

The device does not have a DirectSound driver installed, so it is being emulated through the waveform-audio functions. Performance degradation should be expected.

DSCAPS_PRIMARY16BIT

The device supports primary buffers with 16-bit samples.

DSCAPS_PRIMARY8BIT

The device supports primary buffers with 8-bit samples.

DSCAPS_PRIMARYMONO

The device supports monophonic primary buffers.

DSCAPS_PRIMARYSTEREO

The device supports stereo primary buffers.

DSCAPS_SECONDARY16BIT

The device supports hardware-mixed secondary buffers with 16-bit samples.

DSCAPS_SECONDARY8BIT

The device supports hardware-mixed secondary buffers with 8-bit samples.

DSCAPS_SECONDARYMONO

The device supports hardware-mixed monophonic secondary buffers.

DSCAPS_SECONDARYSTEREO

The device supports hardware-mixed stereo secondary buffers.

dwMinSecondarySampleRate and **dwMaxSecondarySampleRate**

Minimum and maximum sample rate specifications that are supported by this device's hardware secondary sound buffers.

dwPrimaryBuffers

Number of primary buffers supported. This value will always be 1 for this release.

dwMaxHwMixingAllBuffers

Specifies the total number of buffers that can be mixed in hardware.

dwMaxHwMixingStaticBuffers

Specifies the maximum number of static buffers.

dwMaxHwMixingStreamingBuffers

Specifies the maximum number of streaming buffers.

The value for **MaxHwMixingAllBuffers** may be less than the sum of **dwMaxHwMixingStaticBuffers** and **dwMaxHwMixingStreamingBuffers**. Resource trade-offs frequently occur.

dwFreeHwMixingAllBuffers, **dwFreeHwMixingStaticBuffers**, and **dwFreeHwMixingStreamingBuffers**

Description of the free, or unallocated, hardware mixing capabilities of the device. These values can be used by an application to determine whether hardware resources are available for allocation to a secondary sound buffer. Also, by comparing these values to the members that specify maximum mixing capabilities, the resources that are already allocated can be determined.

dwMaxHw3DAllBuffers, **dwMaxHw3DStaticBuffers**, and **dwMaxHw3DStreamingBuffers**

Description of the hardware 3D positional capabilities of the device. These will all be 0 for the first release.

dwFreeHw3DAllBuffers, **dwFreeHw3DStaticBuffers**, and **dwFreeHw3DStreamingBuffers**

Description of the free, or unallocated, hardware 3D positional capabilities of the device. These will all be 0 for the first release.

dwTotalHwMemBytes

Size, in bytes, of the amount of memory on the sound card that stores static sound buffers.

dwFreeHwMemBytes

Size, in bytes, of the free memory on the sound card.

dwMaxContigFreeHwMemBytes

Size, in bytes, of the largest contiguous block of free memory on the sound card.

dwUnlockTranferRateHwBuffers

Description of the rate, in kilobytes per second, at which data can be transferred to hardware static sound buffers (those located in onboard sound memory). This and the number of bytes transferred determines the duration of a call to the IDirectSoundBuffer::Unlock method.

dwPlayCpuOverheadSwBuffers

Description of the processing overhead, as a percentage of the central processing unit, needed to mix software buffers (those located in main system memory). This varies according to the bus type, the processor type, and the clock speed.

The unlock transfer rate for software buffers is 0 because the data does not need to be transferred anywhere. Similarly, the play processing overhead for hardware buffers is 0 because the mixing is done by the sound device.

dwReserved1, **dwReserved2**

These values are reserved. Do not use.

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all IDirectSound and IDirectSoundBuffer methods. For a list of the error codes each method is capable of returning, see the individual method descriptions.

DS_OK

The request completed successfully.

DSERR_ALLOCATED

The request failed because resources, such as a priority level, were already in use by another caller.

DSERR_ALREADYINITIALIZED

The object is already initialized.

DSERR_BADFORMAT

The specified wave format is not supported.

DSERR_BUFFERLOST

The buffer memory has been lost and must be restored.

DSERR_CONTROLUNAVAIL

The control (volume, pan, and so forth) requested by the caller is not available.

DSERR_GENERIC

An undetermined error occurred inside the DirectSound subsystem.

DSERR_INVALIDCALL

This function is not valid for the current state of this object.

DSERR_INVALIDPARAM

An invalid parameter was passed to the returning function.

DSERR_NOAGGREGATION

The object does not support aggregation.

DSERR_NODRIVER

No sound driver is available for use.

DSERR_OTHERAPPHASPRIO

This value is obsolete and is not used.

DSERR_OUTOFMEMORY

The DirectSound subsystem could not allocate sufficient memory to complete the caller's request.

DSERR_PRIOLEVELNEEDED

The caller does not have the priority level required for the function to succeed.

DSERR_UNINITIALIZED

The IDirectSound::Initialize method has not been called or has not been called successfully before other methods were called.

DSERR_UNSUPPORTED

The function called is not supported at this time.

Overview

The Microsoft® DirectPlay™ application programming interface (API) for Windows® 95 is a software interface that simplifies your application's access to communication services. DirectPlay was primarily developed for game communication, but could be used for other applications that require communication independent of the underlying transport, protocol, or online service.

Games are more fun if they can be played against real players, and the personal computer has richer connectivity options than any game platform in history. Instead of forcing you, as a game developer, to deal with the differences that each of these connectivity solutions represents, DirectPlay provides well defined, generalized communication capabilities. DirectPlay shields you from the underlying complexities of diverse connectivity implementations, freeing you to concentrate on producing a great game.

DirectPlay Architecture

DirectPlay uses a simple send/receive communication model to implement a connectivity API tailored to the needs of game play. The DirectPlay architecture is composed of two types of components: DirectPlay itself and the service provider. DirectPlay is provided by Microsoft and presents a common interface to the game. The service provider furnishes medium-specific communication services as requested by DirectPlay. Any organization, including online services, can supply service providers for specialized hardware and communications media. Microsoft includes two service providers with DirectPlay, one for networking and one for modem support.

The DirectPlay interface hides the complexities and unique tasks required to establish an arbitrary communications link inside the DirectPlay service provider implementation. When you design a game for use with DirectPlay, you need only concern yourself with the performance of the communications medium, not whether that medium is provided by a modem, network, or online service.

DirectPlay will dynamically bind to any DirectPlay service provider installed on the user's system. The game interacts with the DirectPlay object. The DirectPlay object interacts with one of the available DirectPlay service providers, and the selected service provider interacts with the transport or protocol.

Globally Unique Identifiers

Each game requires a globally unique identifier (GUID) that it uses to identify itself on the communications medium. These GUIDs, sometimes called UUIDs, can be generated on any computer that has a network card and a copy of Uuidgen.exe, provided as part of the Microsoft Win32® SDK. You create a GUID once while developing the game, and use that GUID throughout the life of the product. There is no need to register this number with Microsoft.

Using DirectPlay

Implement DirectPlay in your application using the following steps:

1. Request that the user of your application select a communication medium for the game.

   You can identify the service providers installed on a personal computer by using the DirectPlayEnumerate function.
2. Create a DirectPlay object based on the selected provider by calling the DirectPlayCreate function and specifying the appropriate service provider GUID.

   Calling the **DirectPlayCreate** function causes DirectPlay to load the library for the service provider selected.
3. Request game information, including preferences, from the user. You can store this information in the **dwUser** member of the DPSESSIONDESC structure.

   If the user wants to start a new game, skip to step six.
4. Enumerate existing sessions (existing games that a user can join) by using the IDirectPlay::EnumSessions method.
5. If the user wants to join a game enumerated by the **IDirectPlay::EnumSessions** method, connect to that game by using the IDirectPlay::Open method and specify the DPOPEN_OPENSESSION flag.
6. If the user wants to start a new game, create a game by using the **IDirectPlay::Open** method and specifying the DPOPEN_CREATESESSION flag.
7. Create a player or players.

   A player's communication capabilities can be determined using the IDirectPlay::GetCaps and IDirectPlay::GetPlayerCaps methods. Other players can be discovered by using the IDirectPlay::EnumPlayers method.
8. Exchange messages among players, including the name server (player ID 0), by using the IDirectPlay::Send and IDirectPlay::Receive methods.

Each player is associated with a friendly name and a formal name that the game can use for tasks such as error reporting and scoring. The game can exchange messages among players by using the unique player ID that is created with the player. The service provider, not DirectPlay, limits the number of players that can participate in a gaming session. In the current implementation, the number of players ranges from 16 for a modem connection to 256 for a network connection.

When using DirectPlay, you are able to define most messages in order to address the particular needs of the game. However, some system messages are defined by DirectPlay. For example, when a player quits or joins a game, that game receives a system message that provides the player's name and the status change that has just occurred. System messages are always sent by the name server, a virtual player whose player ID is 0. System messages start with a 32-bit value that identifies the type of message. Constants that represent system messages begin with 'DPSYS_', and have a corresponding message structure that must be used to interpret them.

Broadcasting a message to all players in the game is simply a matter of sending a message to the name server (that is, to player ID 0). Players receiving a message broadcast in this way see the message as having come from the player who sent it, not from the name server.

DirectPlay does not attempt to provide a general approach for game synchronization; to do so would necessarily impose limitations on the game-playing paradigm. However, the system includes some services that are designed to help you with these tasks. For example, you can specify a notification event when your application creates a player, then use the Microsoft Win32 function **WaitForSingleObject** to find out whether there is a message pending for that player.

Session Management

A DirectPlay session is an instance of a game. The applications you design can use DirectPlay's session-management functions to open or close a communication channel, save a session in the registry, or enumerate past sessions that have been saved in the registry. A game either creates a new session or enumerates existing or previous sessions and finds one to connect to. If a game has saved a session, it could enumerate previous sessions and perhaps reconnect to the saved session. (This is a particularly appropriate scenario in a modem environment, where a saved session would include phone numbers.) Not all DirectPlay service providers will support saving sessions, however, and this functionality is currently only implemented for modem connections.

The IDirectPlay::Open method is used to create new sessions or to connect to existing or saved sessions. A session is described by its corresponding DPSESSIONDESC structure. This structure contains game-specific values and session particulars, such as the name of the session, an optional password for the session, and the number of players to be allowed in the session. After opening a session, you can call the IDirectPlay::GetCaps method to retrieve the speed of the communications link. To save a record of the session in the registry, call the IDirectPlay::SaveSession method. For a modem connection, you can save the current session and later enumerate all of the saved sessions by calling IDirectPlay::EnumSessions and specifying the DPENUMSESSIONS_PREVIOUS flag. Opening one of these saved sessions retrieves the phone number for that session and dials it. When a game session is over, it can be closed with the IDirectPlay::Close method.

Player Management

The applications you design can use DirectPlay's player-management methods to manage the players in a game session. In addition to creating and destroying players, you can enumerate them or retrieve their communication capabilities.

The IDirectPlay::CreatePlayer and IDirectPlay::DestroyPlayer methods create and delete players in a game session. Upon creation, each player is given a friendly name, a formal name, and a DirectPlay player ID. The player ID is used by the game and DirectPlay to route message traffic. The friendly and formal names are not used internally by DirectPlay; instead, you can use them when communicating with the players. The IDirectPlay::GetPlayerName and IDirectPlay::SetPlayerName methods allow your application to work with the friendly and formal names while the game is being played. The IDirectPlay::EnableNewPlayers method enables or disables the addition of new players and can be used to prohibit the creation of new players once a game is in progress.

You can use the IDirectPlay::EnumPlayers method to discover what players are in a current game session and their friendly and formal names. This function is typically called immediately after the IDirectPlay::Open method opens an existing session. The IDirectPlay::GetPlayerCaps method retrieves information about the speed of a player's connection to the session.

Group Management

The group-management methods allow your application to create groups of players in a session. You can then use a single instance of the IDirectPlay::Send method to send messages to an entire group, rather than to one player at a time. Some service providers can send messages to groups more efficiently than they can send them to the individual players, so, in addition to simplifying player management, groups can be used to conserve communication channel bandwidth.

The IDirectPlay::CreateGroup and IDirectPlay::DestroyGroup methods create and delete a group of players. When you create a group, you assign it a friendly and formal name, just as you would when creating a player. The group is initially empty; you can use the IDirectPlay::AddPlayerToGroup and IDirectPlay::DeletePlayerFromGroup methods to control the membership of the group. The state of the IDirectPlay::EnableNewPlayers method does not affect the ability to create groups.

To discover what groups exist, you can call the IDirectPlay::EnumGroups method. To enumerate the players in a group, call the IDirectPlay::EnumGroupPlayers method.

Message Management

The message-management functions help your application route messages between game players. With the exception of a small number of messages that have been defined by the system, the messages can be defined in any way that you require. Messages should not be excessively large, however. You can use the IDirectPlay::Send method to send a message to an individual player, to a group, or to all the players in the session; simply specify a player ID, a group ID, or 0 as the destination.

To retrieve a message from the message queue, use the IDirectPlay::Receive method. This method allows you to specify whether to retrieve the first message in the queue, only the messages to a particular player, or only those from a particular player. You can use the IDirectPlay::GetMessageCount method to retrieve the number of messages waiting for a given player.

Functions

The DirectPlay functions are used to initiate communication through the DirectPlay interface. The DirectPlayCreate function is used to instantiate a DirectPlay object for a particular service provider. The DirectPlayEnumerate function is used to obtain a list of all the DirectPlay service providers installed on the system. This is the mechanism DirectPlay uses to support multiple communication transports and protocols. To utilize protocol-independent communication services, your application need only select a specific service provider and instantiate it.

DirectPlayCreate

```
HRESULT DirectPlayCreate(LPGUID lpGUID,
    LPDIRECTPLAY FAR * lplpDP, IUnknown FAR * pUnkOuter);
```

Creates an instance of a DirectPlay object.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_EXCEPTION DPERR_INVALIDPARAMS
  DPERR_UNAVAILABLE

lpGUID
Address for the GUID that represents the driver to be created.

lplpDP
Address for a pointer to initialize with a valid DirectPlay pointer.

pUnkOuter
Address for the IUnknown interface. This parameter is provided for future compatibility with COM aggregation features. Presently, however, the **DirectPlayCreate** function will return an error if this parameter is anything but NULL.

This function attempts to initialize a DirectPlay object and sets a pointer to it if successful. It is a good idea to call the DirectPlayEnumerate function immediately before initialization in order to determine what types of service providers are available.

DirectPlayEnumerate

```
LT DirectPlayEnumerate(
    LPDPENUMDPCALLBACK lpEnumDPCallback, LPVOID lpContext);
```

Enumerates the DirectPlay service providers installed on the system.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_EXCEPTION DPERR_GENERIC

lpEnumDPCallback
Address for the EnumDPCallback function that will be called with a description of each DirectPlay service provider interface installed in the system.

lpContext
Address for a caller-defined structure that will be passed to the callback function each time the function is invoked.

Callback Functions

Most of the functionality of DirectPlay is provided by the methods of its Component Object Model (COM) interfaces. This section lists the callback functions that are not implemented as part of a COM interface.

EnumDPCallback

```
BOOL EnumDPCallback(LPGUID lpSPGuid,
    LPSTR lpFriendlyName, DWORD dwMajorVersion,
    DWORD dwMinorVersion, LPVOID lpContext);
```

Application-defined callback procedure for the DirectPlayEnumerate function.

- Returns TRUE to continue the enumeration or FALSE to stop it.

lpSPGuid
Address for the unique identifier of the DirectPlay service provider driver.

lpFriendlyName
Address for a string containing the driver description.

dwMajorVersion
Major version number of the driver.

dwMinorVersion
Minor version number of the driver.

lpContext
Address for a caller-defined context.

EnumPlayersCallback

```
BOOL EnumPlayersCallback(DPID dpID,
    LPSTR lpFriendlyName, LPSTR lpFormalName,
    DWORD dwFlags, LPVOID lpContext);
```

Application-defined callback procedure for the IDirectPlay::EnumGroups, IDirectPlay::EnumGroupPlayers, and IDirectPlay::EnumPlayers methods.

- Returns TRUE to continue the enumeration or FALSE to stop it.

dpID
Player ID of the group being enumerated.

lpFriendlyName
Address for the zero-terminated string that contains the friendly name of the group.

lpFormalName
Address for the zero-terminated string that contains the formal name of the group.

dwFlags
Specifies the optional control flags.

DPENUMPLAYERS_GROUP
Specifies that group names and player names should be enumerated.

DPENUMPLAYERS_LOCAL
Specifies that only local player names need to be enumerated.

DPENUMPLAYERS_REMOTE
Specifies that only remote player names need to be enumerated.

lpContext
Address for a caller-defined context.

EnumSessionsCallback

```
BOOL EnumSessionsCallback(LPDPSESSIONDESC lpDPSGameDesc,
    LPVOID lpContext, LPDWORD lpdwTimeOut,
    DWORD dwFlags);
```

Application-defined callback procedure for the IDirectPlay::EnumSessions method.

- Returns TRUE to continue the enumeration or FALSE to stop it.

lpDPSGameDesc

Address for a DPSESSIONDESC structure describing the enumerated session. This parameter will be set to NULL if the enumeration has timed out.

lpContext

Address for a caller-defined context.

lpdwTimeOut

Address for a doubleword containing the current time-out value. This can be reset if you feel that some sessions have yet to respond.

dwFlags

Specifies the optional control flag.

DPESC_TIMEDOUT

The enumeration has timed out. Reset *lpdwTimeOut* and return TRUE to continue, or FALSE to stop the enumeration.

IDirectPlay Interface Method Groups

Applications use the methods of the **IDirectPlay** interface to create DirectPlay objects and work with system-level variables. This interface supports the following methods:

Group management	AddPlayerToGroup CreateGroup DeletePlayerFromGroup DestroyGroup EnumGroupPlayers EnumGroups
IUnknown	AddRef QueryInterface Release
Message management	GetMessageCount Receive Send
Player management	CreatePlayer DestroyPlayer EnableNewPlayers EnumPlayers GetPlayerCaps GetPlayerName SetPlayerName
Session management	Close EnumSessions GetCaps Open SaveSession

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the DirectPlay object without affecting the functionality of the original interface.

IDirectPlay::AddPlayerToGroup

```
HRESULT AddPlayerToGroup(DPID pidGroup, DPID pidPlayer);
```

Adds an existing player to an existing group.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_GENERIC DPERR_INVALIDOBJECT
  DPERR_INVALIDPLAYER

pidGroup
ID of the group to be augmented.

pidPlayer
ID of the player to be added to the group.

A DPMSG_GROUPADD system message is automatically generated to inform other players that the specified player has been added. Groups cannot be added to other groups, but players can be members of multiple groups. The **IDirectPlay::AddPlayerToGroup** method will generate an error if the player is already a member of the group.

IDirectPlay::AddRef

```
ULONG AddRef();
```

Increases the reference count of the DirectPlay object by 1. This method is part of the IUnknown interface inherited by DirectPlay.

- Returns the new reference count of the object.

When the DirectPlay object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirectPlay::Release method to decrease the reference count of the object by 1.

IDirectPlay::Close

```
HRESULT Close();
```

Closes a previously opened communication channel. All locally created players will be destroyed, with corresponding DPMSG_DELETEPLAYER system messages sent to other session participants.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS

IDirectPlay::CreateGroup

```
HRESULT CreateGroup(LPDPID lppidID,
    LPSTR lpGroupFriendlyName, LPSTR lpGroupFormalName);
```

Creates a group of players for a session. A player ID representing the new group will be returned to the caller.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_CANTADDPLAYER DPERR_INVALIDOBJECT
  DPERR_INVALIDPARAMS DPERR_OUTOFMEMORY

lppidID
Address for the DPID that will hold the DirectPlay player ID.

lpGroupFriendlyName
Address for the zero-terminated string that contains the friendly name of the group.

lpGroupFormalName
Address for the zero-terminated string that contains the formal name of the group.

Messages sent to a player ID designating this group will be sent to all members of the group. The *lpGroupFriendlyName* and *lpGroupFormalName* parameters are provided for players' convenience only; they are not used internally and do not need to be unique. However, player IDs assigned by DirectPlay will always be unique within a session. Groups are recognized as players for the session player count; if you have a four player game with four existing players, calls to **IDirectPlay::CreateGroup** will fail. The state of the IDirectPlay::EnableNewPlayers method does not affect your application's ability to create groups.

Upon successful completion, this method sends a DPMSG_ADDPLAYER system message to all of the other players in the game announcing that a new group has been created.

IDirectPlay::CreatePlayer

```
HRESULT CreatePlayer(LPDPID lppidID,
    LPSTR lpPlayerFriendlyName,
    LPSTR lpPlayerFormalName, LPHANDLE lpEvent);
```

Creates a player for the current game session.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_CANTADDPLAYER DPERR_CANTCREATEPLAYER
  DPERR_GENERIC DPERR_INVALIDOBJECT
  DPERR_INVALIDPARAMS DPERR_NOCONNECTION

lppidID
Address for the DPID that will hold the DirectPlay player ID.

lpPlayerFriendlyName
Address for the zero-terminated string that contains the friendly name of the player.

lpPlayerFormalName
Address for the zero-terminated string that contains the formal name of the player.

lpEvent
Pointer to an event that will be triggered when a message addressed to this player is received.

A single process can have multiple players that communicate through a DirectPlay object with any number of other players running on multiple computers. The player ID returned to the caller will be used internally to direct the player's message traffic and manage the player. The *lpPlayerFriendlyName* and *lpPlayerFormalName* parameters are provided for players' convenience only; they are not used internally and need not be unique. Player IDs assigned by DirectPlay will always be unique within the session.

Upon successful completion, this method sends a DPMSG_ADDPLAYER system message to all of the other players in the game session announcing that a new player has joined the session. The newly created player can use the IDirectPlay::EnumPlayers method to find out who else is in the game session.

It is highly recommended that an application provide a non-NULL *lpEvent* and use this event for synchronization. After creation of a player, use *WaitForSingleObject(*lpEvent, dwTimeout = 0)* from Win32 to determine if a player has messages (the return value will be WAIT_TIMEOUT if there are not any waiting messages) or use a different time-out to wait for a message to come in. It is inefficient to loop on the IDirectPlay::Receive method.

IDirectPlay::DeletePlayerFromGroup

```
HRESULT DeletePlayerFromGroup(DPID pidGroup,
    DPID pidPlayer);
```

Removes a player from a group.

- Returns DP_OK if successful, or one of the following error messages otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPLAYER

pidGroup
Player ID of the group to be adjusted.

pidPlayer
ID of the player to be removed from the group.

A DPMSG_GROUPDELETE system message is automatically generated to inform the other players of this change.

IDirectPlay::DestroyGroup

```
HRESULT DestroyGroup(DPID pidID);
```

Deletes a group from the session. The player ID belonging to this group will not be reused during the current session.

- Returns DP_OK if successful, or one of the following error messages otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPLAYER

pidID

Player ID of the group being removed from the game.

It is not necessary to empty a group before deleting it. The individual players belonging to the group are not destroyed; however, they will be notified by a DPMSG_DELETEPLAYER system message that the group has been removed from the session.

IDirectPlay::DestroyPlayer

```
HRESULT DestroyPlayer(DPID pidID);
```

Deletes a player from the game session, removes any pending messages destined for that player from the message queue, and removes the player from any groups to which it belonged. The player ID will not be reused during the current session.

- Returns DP_OK if successful, or one of the following error messages otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPLAYER

pidID

ID of the player that is being removed from the game.

Calling this method automatically sends a DPMSG_DELETEPLAYER system message to all other players, informing them that this player has been removed from the session.

IDirectPlay::EnableNewPlayers

```
HRESULT EnableNewPlayers(BOOL bEnable);
```

Enables or disables the creation of new players.

- Returns DP_OK if successful, or DPERR_INVALIDOBJECT otherwise.

bEnable

If TRUE (the default condition for a session), new players can be created unless the session has reached its maximum capacity. If FALSE, any attempt to create a new player will return an error.

This method does not affect your ability to create groups. Typically, new players and groups can be added to a session until the session's player limit has been reached. This method can override this behavior if, for example, a session is in progress and new players are not desired. The IDirectPlay::EnumSessions method will not enumerate sessions where **IDirectPlay::EnableNewPlayers** has been set to FALSE unless the DPENUMSESSIONS_ALL flag is used.

IDirectPlay::EnumGroupPlayers

```
HRESULT EnumGroupPlayers(DPID pidGroupPID,
    LPDPENUMPLAYERSCALLBACK lpEnumPlayersCallback,
    LPVOID lpContext, DWORD dwFlags);
```

Enumerates all of the players of a particular group existing in the current session.

- Returns DP_OK if successful, or one of the following error messages otherwise:
  DPERR_EXCEPTION DPERR_INVALIDFLAGS
  DPERR_INVALIDOBJECT DPERR_INVALIDPLAYER

pidGroupPID
Player ID of the group to be enumerated.

lpEnumPlayersCallback
Address for the EnumPlayersCallback function to be called for every player in the group.

lpContext
Address for a caller-defined context that is passed to each enumeration callback.

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

IDirectPlay::EnumGroups

```
HRESULT EnumGroups(DWORD dwSessionID,
    LPDPENUMPLAYERSCALLBACK lpEnumPlayersCallback,
    LPVOID lpContext, DWORD dwFlags);
```

Enumerates the groups in a session.

- Returns DP_OK if successful, or one of the following error messages otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS
  DPERR_UNSUPPORTED

dwSessionID

Session of interest. Do not use unless the DPENUMPLAYERS_SESSION flag is specified.

lpEnumPlayersCallback

Address of the EnumPlayersCallback function to be called for every group in the session.

lpContext

Address for a caller-defined context that is passed to each enumeration callback.

dwFlags

Specifies the optional control flag.

DPENUMPLAYERS_SESSION

Requests the name server for the specified session supply its group list.

By default, this method will enumerate using the local player list for the current session. The DPENUMPLAYERS_SESSION flag can be used, along with a session ID, to request that a session's name server provide its list for enumeration. This method cannot be called from within an IDirectPlay::EnumSessions enumeration. Furthermore, use of the DPENUMPLAYERS_SESSION flag with this method must occur after the **IDirectPlay::EnumSessions** method has been called, and before any attempt to open or close has been made.

IDirectPlay::EnumPlayers

```
HRESULT EnumPlayers(DWORD dwSessionId,
    LPDPENUMPLAYERSCALLBACK lpEnumPlayersCallback,
    LPVOID lpContext, DWORD dwFlags);
```

Enumerates the players in a session. Groups can also be included in the enumeration by using the DPENUMPLAYERS_GROUP flag.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_EXCEPTION DPERR_GENERIC
  DPERR_INVALIDOBJECT DPERR_UNSUPPORTED

dwSessionID

Session of interest. Not used unless the DPENUMPLAYERS_SESSION flag is specified.

lpEnumPlayersCallback

Address for the EnumPlayersCallback function that will be called for every player in the session.

lpContext

Address for a caller-defined context that is passed to each enumeration callback.

dwFlags

Specifies the control flags.

DPENUMPLAYERS_GROUP

Includes groups in the enumeration of players.

DPENUMPLAYERS_PREVIOUS

Enumerates players previously stored in the registry. Not yet supported.

DPENUMPLAYERS_SESSION

Requests that the name server supply its group list for the specified session.

This method is often called immediately after the DirectPlay object is opened, and will enumerate using the local player list for the current session by default. The **IDirectPlay::EnumPlayers** method may be called after the IDirectPlay::EnumSessions method in order to obtain a list of players in a particular session. **IDirectPlay::EnumPlayers** uses the DPENUMPLAYERS_SESSION flag and the session ID returned from the **IDirectPlay::EnumSessions** method to obtain this list. However, it cannot be called from within an **IDirectPlay::EnumSessions** enumeration. The use of the DPENUMPLAYERS_SESSION flag with this method must occur after the **IDirectPlay::EnumSessions** method has been called and before either the IDirectPlay::Close or the IDirectPlay::Open method has been called.

IDirectPlay::EnumSessions

```
HRESULT EnumSessions(LPDPSESSIONDESC lpSDesc,
    DWORD dwTimeout,
    LPDPENUMSESSIONSCALLBACK lpEnumSessionsCallback,
    LPVOID lpContext, DWORD dwFlags);
```

Enumerates the game sessions connected to this DirectPlay object.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS
  DPERR_NOSESSIONS

lpSDesc

Address for a DPSESSIONDESC structure describing the sessions to be enumerated. If a list of all connected sessions, regardless of GUID, is desired, then the **guidSession** member in the **DPSESSIONDESC** structure should be set to NULL. If the DPENUMSESSIONS_AVAILABLE flag is going to be used with a password, then the **szPassword** member should be set accordingly.

dwTimeout

Total amount of time, in milliseconds, that DirectPlay will allow for the enumeration to complete (not the time between each enumeration).

lpEnumSessionsCallback

Address for the EnumSessionsCallback function to be called for each DirectPlay session responding.

lpContext

Address for a user-defined context that is passed to each enumeration callback.

dwFlags

Specifies the optional control flags.

DPENUMSESSIONS_AVAILABLE

Enumerates all sessions with a matching password (if provided), opens player slots, and sets the IDirectPlay::EnableNewPlayers method to TRUE.

DPENUMSESSIONS_PREVIOUS

Enumerates sessions previously stored in the registry.

This method is usually called immediately after the DirectPlay object is instantiated; it cannot be called while a game is connected to a session or after a game has created a session.
IDirectPlay::EnumSessions broadcasts an enumeration request and collects replies from the DirectPlay objects that respond. The amount of time DirectPlay spends listening for these replies is controlled by the *dwTimeout* parameter. When this time interval has expired, your callback will be notified using the DPESC_TIMEDOUT flag, and a NULL value will be passed for the *lpDPSGameDesc* parameter. At this point, you may choose to continue the enumeration by setting *dwTimeOut* to a new value and returning TRUE, or returning FALSE to cancel the enumeration.

IDirectPlay::GetCaps

```
HRESULT GetCaps(LPDPCAPS lpDPCaps);
```

Obtains the capabilities of this DirectPlay object.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS

lpDPCaps
Address for a DPCAPS structure that will be filled with the capabilities of the DirectPlay object.

IDirectPlay::GetMessageCount

```
HRESULT GetMessageCount(DPID pidID, LPDWORD lpdwCount);
```

Queries the name server for the number of messages queued for a specific local player.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS
  DPERR_INVALIDPLAYER

pidID
Player ID for which the message count is requested. The player must be local.

lpdwCount
Address for a doubleword that will be set to the message count.

IDirectPlay::GetPlayerCaps

```
HRESULT GetPlayerCaps(DPID pidID, LPDPCAPS lpDPPlayerCaps);
```

Obtains the capabilities of the player's connection by querying the DirectPlay object.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS
  DPERR_INVALIDPLAYER

pidID
Player ID for which the communication capabilities are requested.

lpDPPlayerCaps
Address for a DPCAPS structure that will be filled with the communication capabilities of the specified player on this DirectPlay object.

This method is needed because communicating with some players may be slower than communicating with others.

IDirectPlay::GetPlayerName

```
HRESULT GetPlayerName(DPID pidID,
    LPSTR lpPlayerFriendlyName,
    LPDWORD lpdwFriendlyNameLength,
    LPSTR lpPlayerFormalName,
    LPDWORD lpdwFormalNameLength);
```

Obtains the player's friendly and formal names from the name server.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_BUFFERTOOSMALL DPERR_INVALIDOBJECT
  DPERR_INVALIDPARAMS DPERR_INVALIDPLAYER

pidID
Player ID for which the player names are being requested.

lpPlayerFriendlyName
Address for the buffer to be filled with the player's friendly name. Set to NULL if you are only looking for the size of the friendly name or are only looking for the formal name.

lpdwFriendlyNameLength
Address for a doubleword that either contains the length of the buffer pointed to by the *lpPlayerFriendlyName* field or will be filled with the length needed for the buffer. Set to NULL if you are only interested in the formal name.

lpPlayerFormalName
Address for the buffer to be filled with the player's formal name. Set to NULL if you are only looking for the size of the formal name or are only looking for the friendly name.

lpdwFormalNameLength
Address for a doubleword that either contains the length of the buffer pointed to by the *lpPlayerFormalName* parameter or will be filled with the length needed for the buffer. Set to NULL if you are only interested in the friendly name.

If just one of the names is required, you can set the pair of pointers to the other name to NULL. If the length of the names needs to be determined, the pointers to the lengths must be valid; however, they can point to zeros or you can set the pointers to the friendly and formal names to NULL.

If the supplied buffer is not long enough to hold one of the names, an error code will be returned and the corresponding buffer length will be adjusted to be the size of the buffer needed.

IDirectPlay::Initialize

```
HRESULT Initialize(LPGUID lpGUID);
```

Provided to comply with the Common Object Model (COM) protocol.

- Returns DPERR_ALREADYINITIALIZED.

lpGUID

Address for the GUID used as the interface identifier.

Since the DirectPlay object is initialized when it is created, calling this method will always result in the DPERR_ALREADYINITIALIZED return value.

IDirectPlay::Open

```
HRESULT Open(LPDPSESSIONDESC lpSDesc);
```

Establishes the communication link between DirectPlay and a service provider.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_ACTIVEPLAYERS DPERR_GENERIC
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS
  DPERR_UNAVAILABLE DPERR_UNSUPPORTED
  DPERR_USERCANCEL

lpSDesc
Address for the DPSESSIONDESC structure describing the session to be connected to or created.

In a modem environment, calling this method is equivalent to actually dialing the phone. This prompts the required user interface to configure the communications protocol that will be invoked with the DirectPlay object. In the case of the serial modem service provider supplied with DirectPlay, the user is prompted for dialing information. If the user cancels the dialing process, **IDirectPlay::Open** will return a DPERR_USERCANCEL error. This method will also return an error if any local players exist when it is called.

IDirectPlay::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID * ppvObj);
```

Determines if the DirectPlay object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by DirectPlay.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS

riid
Reference identifier of the interface being requested.

ppvObj
Address for a pointer to be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirectPlay::QueryInterface** method allows DirectPlay objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirectPlay::Receive

```
HRESULT Receive(LPDPID lppidFrom, LPDPID lppidTo,
    DWORD dwFlags, LPVOID lpvBuffer, LPDWORD lpdwSize);
```

Retrieves a message from the message queue.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_BUFFERTOOSMALL DPERR_GENERIC
  DPERR_INVALIDOBJECT DPERR_INVALIDPARAMS
  DPERR_NOMESSAGES

lppidFrom
Address for a DPID structure to be filled with the sender's player ID.

lppidTo
Address for a DPID structure to be filled with the receiver's player ID.

dwFlags
Specifies the optional control flags.

DPRECEIVE_ALL
Returns the first available message. This is the default.

DPRECEIVE_FROMPLAYER
Returns the first message from the player ID that the *lppidFrom* parameter points to.

DPRECEIVE_PEEK
Returns a message as specified by the other flags, but does not remove it from the message queue.

DPRECEIVE_TOPLAYER
Returns the first message intended for the player ID that the *lppidTo* parameter points to. System messages are addressed to player ID 0.

lpvBuffer
Address for the message buffer. If this buffer is not long enough to hold the message, an error will be returned and *lpdwSize* will be filled with the size of message buffer needed.

lpdwSize
Address for the doubleword that specifies the length of the message buffer.

Any message received from player ID 0 is a system message from the name server. A message sent to the name server to broadcast to all players still appears to be from the sender. Both the DPRECEIVE_TOPLAYER and DPRECEIVE_FROMPLAYER flags can be specified, in which case **IDirectPlay::Receive** will return whichever message is encountered first.

IDirectPlay::Release

```
ULONG Release();
```

Decreases the reference count of the DirectPlay object by 1. This method is part of the IUnknown interface inherited by DirectPlay.

- Returns the new reference count of the object.

The DirectPlay object deallocates itself when its reference count reaches 0. Use the IDirectPlay::AddRef method to increase the reference count of the object by 1.

IDirectPlay::SaveSession

```
HRESULT SaveSession();
```

Saves the current session in the registry.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_GENERIC DPERR_INVALIDOBJECT
  DPERR_INVALIDPARAMS DPERR_OUTOFMEMORY
  DPERR_UNSUPPORTED

The functionality of this method is dependent on the service provider, which will save enough transport-specific information in the registry to restore the connection. **IDirectPlay::SaveSession** is unsupported in the TCP and IPX service providers. In the serial modem service provider, **IDirectPlay::SaveSession** functions only for the client session (the one that dials), in which case it will save the phone number in the registry for future use.

IDirectPlay::Send

```
HRESULT Send(DPID pidFrom, DPID pidTo, DWORD dwFlags,
    LPVOID lpvBuffer, DWORD dwBuffSize);
```

Sends messages to other players, to other groups of players, or to all players in the session.

- Returns DP_OK if successful, the number of messages waiting for transmission in DirectPlay's internal queue, or one of the following error values:

DPERR_BUSY	DPERR_INVALIDOBJECT
DPERR_INVALIDPLAYER	DPERR_SENDTOOBIG

pidFrom

ID of the sending player.

pidTo

ID of the receiving player.

dwFlags

Indicates how the message should be sent. Not all options may be supported by a particular service provider.

DPSEND_GUARANTEE

Sends the message using a reliable method. Retries until the message is received or until the DirectPlay time-out occurs.

DPSEND_HIGHPRIORITY

Sends the message as a HIGHPRIORITY message.

DPSEND_TRYONCE

Sends the message without error detection and without retry options enabled.

lpvBuffer

Address for the message being sent.

dwBuffSize

Length of the message being sent.

To send a message to another player, specify the appropriate player ID. To send a message to a group of players, send the message to the player ID assigned to the previously created group. To send messages to the entire session, specify the player ID 0, which always represents the name server. **IDirectPlay::Send** will either return one of the values listed above or the number of messages currently queued for transmission. If the internal queue fills to the limit specified by the **dwMaxQueueSize** member of the DPCAPS structure, an error will be generated and the message will not be added to the queue. The **IDirectPlay::Send** method cannot be used inside an IDirectDrawSurface::Lock / IDirectDrawSurface::Unlock or IDirectDrawSurface::GetDC / IDirectDrawSurface::ReleaseDC method pair.

IDirectPlay::SetPlayerName

```
HRESULT SetPlayerName(DPID pidID,
    LPSTR lpPlayerFriendlyName, LPSTR lpPlayerFormalName);
```

Changes the player's friendly and formal names.

- Returns DP_OK if successful, or one of the following error values otherwise:
  DPERR_INVALIDPLAYER DPERR_INVALIDOBJECT

pidID
ID for which the player name is being requested.

lpPlayerFriendlyName
Address to a string containing the player's new friendly name.

lpPlayerFormalName
Address to a string containing the player's new formal name.

This method cannot be used to change the names of a group.

Structures

DirectPlay structures must have their size fields, where present, properly set before calling DirectPlay functions or an error will result.

Data structures	DPCAPS DPSESSIONDESC
Message structures	DPMSG_ADDPLAYER DPMSG_DELETEPLAYER DPMSG_GENERIC DPMSG_GROUPADD DPMSG_GROUPDELETE

DPCAPS

```
typedef struct {
    DWORD    dwSize;
    DWORD    dwFlags;
    DWORD    dwMaxBufferSize;
    DWORD    dwMaxQueueSize;
    DWORD    dwMaxPlayers;
    DWORD    dwHundredBaud;
    DWORD    dwLatency;
} DPCAPS;
```

Contains the capabilities of a DirectPlay object after a call to the IDirectPlay::GetCaps method. This structure is read-only.

dwSize
Size of this structure, in bytes. Must be initialized before the structure is used.

dwFlags
Specifies the optional control flags.

DPCAPS_GUARANTEED
Supports verification of received messages. Retransmits the message, if necessary.

DPCAPS_NAMESERVER
The computer represented by the calling application is the name server.

DPCAPS_NAMESERVICE
A name server is supported.

dwMaxBufferSize
Maximum buffer size for this DirectPlay object.

dwMaxQueueSize
Maximum queue size for this DirectPlay object.

dwMaxPlayers
Maximum number of players supported in a session.

dwHundredBaud
Baud rate specified in hundredths. For example, 2400 baud is represented by the value 24.

dwLatency
Latency estimate per service provider, in milliseconds. If this value is 0, DirectPlay cannot provide an estimate. Accuracy for some service providers rests on application-to-application testing, taking into consideration the average message size.

DPSESSIONDESC

```
typedef struct {
    DWORD    dwSize;
    GUID     guidSession;
    DWORD    dwSession;
    DWORD    dwMaxPlayers;
    DWORD    dwCurrentPlayers;
    DWORD    dwFlags;
    char     szSessionName[DPSESSIONNAMELEN];
    char     szUserField[DPUSERRESERVED];
    DWORD    dwReserved1;
    char     szPassword[DPPASSWORDLEN];
    DWORD    dwReserved2;
    DWORD    dwUser1;
    DWORD    dwUser2;
    DWORD    dwUser3;
    DWORD    dwUser4;
} DPSESSIONDESC;
typedef DPSESSIONDESC FAR *LPDPSESSIONDESC;
```

Contains a description of the capabilities of a DirectPlay session.

dwSize
Size of this structure, in bytes. Must be initialized before the structure is used.

guidSession
Globally unique identifier (GUID) for the game. It identifies the game so that DirectPlay connects only to other machines playing the same game.

dwSession
Session identifier of the session that has been created or opened.

dwMaxPlayers
Maximum number of players and groups allowed in this session. This member is ignored if the application is not creating a new session.

dwCurrentPlayers
Current players and groups in the session.

dwFlags
Specifies one of the control flags:

DPOPEN_CREATESESSION
Creates a new session described by the **DPSESSIONDESC** structure.

DPOPEN_OPENSESSION
Opens the session identified by the **dwSession** member.

DPENUMSESSIONS_ALL
Enumerates all active sessions connected to this DirectPlay object, regardless of their occupancy, passwords, or the IDirectPlay::EnableNewPlayers method's status.

szSessionName
String containing the name of the session.

szUserField
String containing user data.

dwReserved1
Reserved for future use.

szPassword
String containing the optional password that, once set up, is required to join this session.

dwReserved2
Reserved for future use.

dwUser1, **dwUser2**, **dwUser3**, **dwUser4**
User-specific data for the game or session.

System Messages

Messages returned by the IDirectPlay::Receive method from player ID 0 are system messages. All system messages begin with a doubleword **dwType**. You can cast the buffer returned by the **IDirectPlay::Receive** method to a generic message (DPMSG_GENERIC) and switch on the **dwType** element, which will have a value equal to one of the DPSYS_* messages (DPSYS_ADDPLAYER, and so on).

Your application should be prepared to handle the following system messages:

Value of dwType	Message structure
DPSYS_ADDPLAYER	DPMSG_ADDPLAYER
DPSYS_ADDPLAYERTOGROUP	DPMSG_GROUPADD
DPSYS_DELETEGROUP	DPMSG_DELETEPLAYER
DPSYS_DELETEPLAYER	DPMSG_DELETEPLAYER
DPSYS_DELETEPLAYERFROMGRP	DPMSG_GROUPDELETE
DPSYS_SESSIONLOST	DPMSG_GENERIC

DPMSG_ADDPLAYER

```
typedef struct{
    DWORD    dwType;
    DWORD    dwPlayerType;
    DPID     dpId;
    char     szLongName[DPLONGNAMELEN];
    char     szShortName[DPSHORTNAMELEN];
    DWORD    dwCurrentPlayers;
} DPMSG_ADDPLAYER;
```

Contains information for the DPSYS_ADDPLAYER system message. The system sends this message when players and groups are added to a session.

dwType
Identifier for the message.

dwPlayerType
Indicates whether a player or a group was added. TRUE indicates a player was added; FALSE indicates a group was added.

dpId
ID for a player or group.

szLongName
Formal name for player or group.

szShortName
Friendly name for player or group.

dwCurrentPlayers
Number of players in the session.

DPMSG_DELETEPLAYER

```
typedef struct{
    DWORD    dwType;
    DPID     dpId;
} DPMSG_DELETEPLAYER;
```

Contains information for the DPSYS_DELETEPLAYER and DPSYS_DELETEGROUP system messages. The system sends these messages when players and groups are deleted from a session.

dwType

Identifier for the message.

dpId

ID of a player or group that has been deleted.

DPMSG_GENERIC

```
typedef struct{
    DWORD    dwType;
} DPMSG_GENERIC;
```

This structure is provided for message processing.

dwType

Identifier for the message.

Your application can first cast the unknown message to the DPMSG_GENERIC type, then perform further processing based on the value of **dwType**. One other system message, DPSYS_SESSIONLOST, also uses this structure.

DPMSG_GROUPADD

```
typedef struct{
    DWORD    dwType;
    DPID     dpIdGroup;
    DPID     dpIdPlayer;
} DPMSG_GROUPADD;
```

Contains information for the DPSYS_ADDPLAYERTOGROUP and DPSYS_DELETEPLAYERFROMGRP system messages. The system sends these messages when players are added to and deleted from a group, respectively.

dwType

Identifier for the message.

dpIdGroup

ID of the group to which the player was added or deleted.

dpIdPlayer

ID of the player that was added to or deleted from the specified group.

DPMSG_GROUPDELETE

```
typedef DPMSG_GROUPADD DMSG_GROUPDELETE;
```

Contains information for the DPSYS_DELETEPLAYERFROMGRP message. For a description of the structure members, see the DPMSG_GROUPADD structure.

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all IDirectPlay methods. For a list of the error codes each method is capable of returning, see the individual method descriptions.

DP_OK

The request completed successfully.

DPERR_ACCESSDENIED

The session is full or an incorrect password was supplied.

DPERR_ACTIVEPLAYERS

The requested operation cannot be performed because there are existing active players.

DPERR_ALREADYINITIALIZED

This object is already initialized.

DPERR_BUFFERTOOSMALL

The supplied buffer is not large enough to contain the requested data.

DPERR_BUSY

The DirectPlay message queue is full.

DPERR_CANTADDPLAYER

The player cannot be added to the session.

DPERR_CANTCREATEPLAYER

A new player cannot be created.

DPERR_CAPSNOTAVAILABLEYET

The capabilities of the DirectPlay object have not been determined yet. This error will occur if the DirectPlay object is implemented on a connectivity solution that requires polling to determine available bandwidth and latency.

DPERR_EXCEPTION

An exception occurred when processing the request.

DPERR_GENERIC

An undefined error condition occurred.

DPERR_INVALIDFLAGS

The flags passed to this function are invalid.

DPERR_INVALIDOBJECT

The DirectPlay object pointer is invalid.

DPERR_INVALIDPARAMS

One or more of the parameters passed to the function are invalid.

DPERR_INVALIDPLAYER

The player ID is not recognized as a valid player ID for this game session.

DPERR_NOCAPS

The communication link underneath DirectPlay is not capable of this function.

DPERR_NOCONNECTION

No communication link was established.

DPERR_NOMESSAGES

There are no messages to be received.

DPERR_NONAMESERVERFOUND

No name server could be found or created. A name server must exist in order to create a player.

DPERR_NOPLAYERS

There are no active players in the session.

DPERR_NOSESSIONS

There are no existing sessions for this game.

DPERR_OUTOFMEMORY

There is insufficient memory to perform the requested operation.

DPERR_SENDTOOBIG

The message buffer passed to the IDirectPlay::Send method is larger than allowed.

DPERR_TIMEOUT

The operation could not be completed in the specified time.

DPERR_UNAVAILABLE

The requested service provider or session is not available.

DPERR_UNSUPPORTED

The function is not available in this implementation.

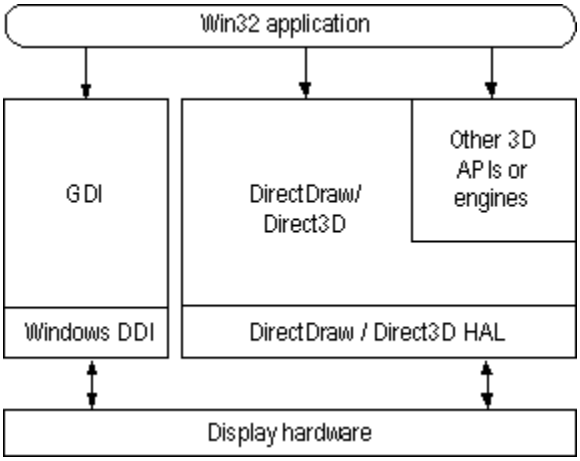
DPERR_USERCANCEL

The user canceled the connection process during a call to the IDirectPlay::Open method.

Microsoft's 3D-Graphics Solutions

The Microsoft family of advanced 3D-graphics solutions includes the Direct3D™ and OpenGL® application programming interfaces (APIs).

The relationship between the Windows graphics systems, applications written with the Win32® API, the rest of the Windows system, and the hardware is shown in the following illustration.



Direct3D

Direct3D is Microsoft's high-speed 3D software solution. It gives you the ability to create high-performance, real-time 3D applications for typical desktop computers. The system was designed for speed and requires little memory.

Direct3D is implemented in two distinctly different modes: Retained Mode, a high-level API in which the application retains the graphics data, and Immediate Mode, a low-level API in which the application explicitly streams the data out to an execute buffer.

Retained Mode

Direct3D Retained Mode API is designed for manipulating 3D objects and managing 3D scenes. Retained Mode makes it easy to add 3D capabilities to existing Windows-based applications or to create new 3D applications. Its built-in geometry engine supports advanced capabilities like key frame animation and frees you from creating object databases and managing the internal structures of objects. In other words, after using a single call to load a predefined 3D object, the application can use simple methods from the API to manipulate the object in the scene in real-time without having to work explicitly with any of the internal geometry.

Retained Mode is built on top of Immediate Mode and is tightly coupled with the DirectDraw™ application programming interface (API). Microsoft will incorporate Retained Mode into a future version of Windows. For more information, see DirectDraw and Introduction to Direct3D Retained-Mode Objects.

Immediate Mode

Direct3D Immediate Mode is Microsoft's low-level 3D API. It allows you to port games and other high-performance multimedia applications to the Windows operating system.

Immediate Mode is a thin layer above real-time 3D accelerator hardware that gives you access to the features of that hardware. It also offers optimal software rendering for some hardware features that are not present. Immediate Mode gives you the flexibility to exploit your own rendering and scene management technologies. It is a device-independent way for applications to communicate with accelerator hardware at a low level, enabling maximum performance.

Unlike Retained Mode, Immediate Mode does not provide a geometry engine; applications that use Immediate Mode must provide their own object and scene management. Therefore, you should be knowledgeable in 3D graphics programming to use Immediate Mode effectively.

Direct3D is based on the OLE Component Object Model (COM), and is tightly integrated with DirectDraw. Microsoft will incorporate Direct3D into a future version of Windows. For more information, see Introduction to Direct3D Immediate-Mode Objects.

Hardware Abstraction and Emulation

The Direct3D API (like the rest of the DirectX API) is built on top of a hardware-abstraction layer (HAL), which insulates you from device-specific dependencies present in the hardware. A companion piece to the Direct3D HAL is the hardware-emulation layer (HEL). The Direct3D HEL provides software-based emulation of features that are not present in hardware. The combination of these hardware abstraction and emulation layers ensures that the services of the API are always available to you.

The Direct3D HAL is tightly integrated with the DirectDraw HAL and the GDI display driver, giving hardware manufacturers a single, consistent interface to Microsoft graphics APIs, and a long-term, unified driver model for accelerated 3D. Hardware manufacturers need to write only a single driver to accelerate Direct3D, DirectDraw, GDI, and OpenGL. Hardware can accelerate all or part of the 3D graphics rendering pipeline, including geometry transformations, 3D clipping and lighting, and rasterization. The Direct3D HAL has been designed to accommodate future graphics accelerators in addition to those available today.

DirectDraw

DirectDraw provides the fastest way to display graphics on-screen. It is the Windows composition engine for 2D graphics, 3D graphics, and video. DirectDraw composes and moves images very quickly and can employ page flipping to enable smooth animation. This combination of capabilities allows you to create high-speed games and multimedia applications and to port existing titles to Windows quickly and easily. DirectDraw is also the composition engine for all of Microsoft's newest graphics subsystems. You can use it to quickly integrate images generated by Windows GDI, Direct3D, ActiveMovie™, and OpenGL.

DirectDraw is a thin layer above the display hardware that enables you to easily take advantage of the powerful composition abilities of graphics accelerators designed for Windows, including high-speed blitting, interpolated stretching, and overlays. It also supports color space conversion, allowing accelerated video playback. Like Direct3D, DirectDraw is a device-independent way for applications to communicate with hardware. Under MS-DOS, you had to optimize code for each target device. With DirectDraw, however, you can attain high performance consistently across all of the hardware that accelerates DirectDraw.

DirectDraw is a COM-based API. Microsoft will incorporate DirectDraw into a future version of Windows. For more information, see About DirectDraw.

OpenGL

OpenGL is a precise 3D technology used for high-end CAD/CAM, modeling and animation, simulations, scientific visualization, and other exacting 3D-image rendering. It is provided with Windows NT and is available to you for use with Windows 95. Putting OpenGL on Windows 95 means that you can run Win32 OpenGL applications on any Win32 workstation. OpenGL currently takes advantage of any high-end hardware that has OpenGL functionality, using a client driver model designed for OpenGL. A future release of OpenGL will be able to take advantage of lower-priced 3D hardware (provided that it supports the precision conformance requirements of OpenGL) through the Direct3D API, providing a hardware solution that complements Direct3D.

The Direct3D Vision

Direct3D is designed to enable world-class game and interactive 3D graphics on a computer running Windows. Its mission is to provide device-dependent access to 3D video-display hardware in a device-independent manner. The application does not need to implement the specific calling procedures required to draw texture-mapped, perspective-corrected, or alpha-blended 3D primitives on a particular piece of hardware. Simply put, Direct3D is a drawing interface for 3D hardware. It integrates tightly with DirectDraw as its buffer-management system, allowing DirectDraw surfaces to be used both as 3D rendering targets and as source texture maps. This allows for hardware-decompressed motion-video mapping, hardware 3D rendering in 2D overlay planes, or even sprites, for example.

Direct3D is designed to set a standard for hardware acceleration—including geometry transformations, 3D clipping, and lighting. Direct3D provides a highly optimized, software-only implementation of the full 3D rendering pipeline. Any part or all of this pipeline can be replaced at any stage by accelerating hardware. This allows you to write applications now that will be able to use better hardware acceleration as new hardware is developed.

Direct3D is tightly integrated with DirectDraw. The DirectDraw driver COM interfaces and the Direct3D driver COM interface both allow you to communicate with the same underlying object. For more information about the integration of Direct3D and DirectDraw, see Direct3D Integration with DirectDraw. For information about DirectDraw's support the 3D surfaces, see Support for 3D Surfaces.

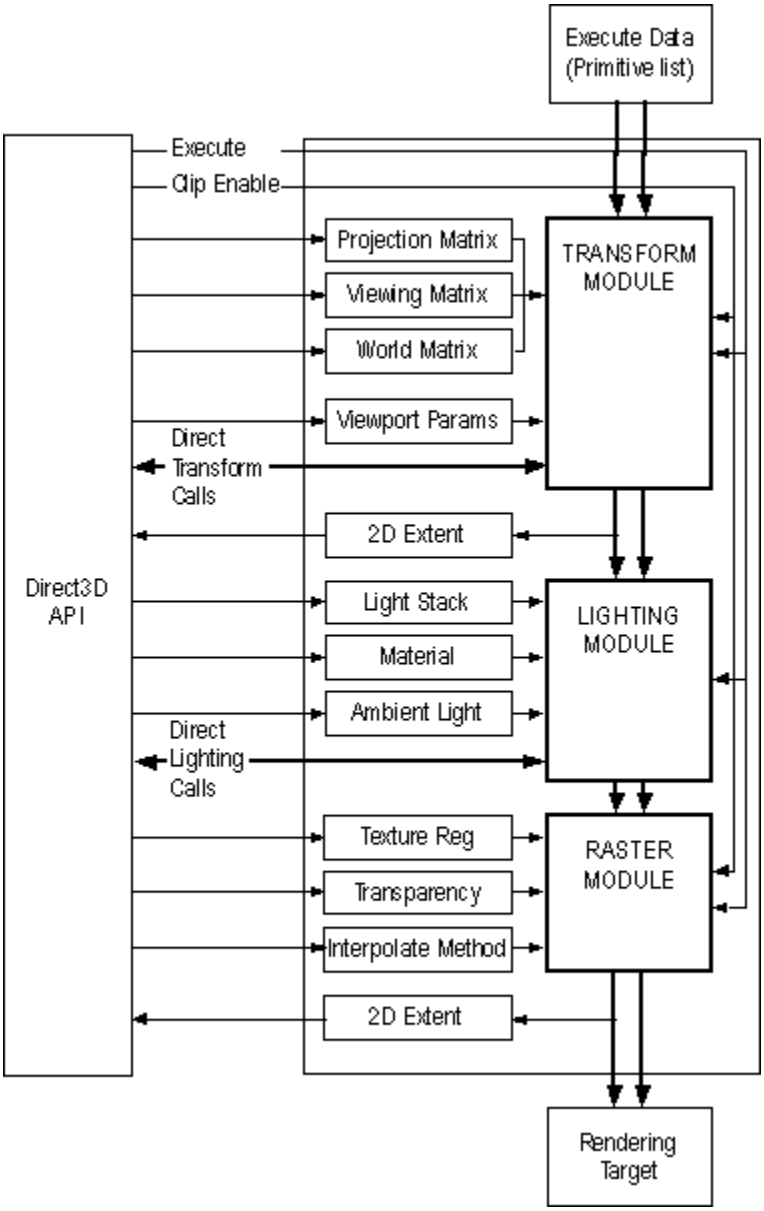
Much of the information in this section describes implementation details of Direct3D that do not directly apply to those of you who use the Retained-Mode interface. If you use the Immediate-Mode interface, however, you'll need a good understanding of these implementation details. Those of you who use the Retained-Mode interface will also benefit from a good theoretical grounding in the system architecture.

Rendering Engine

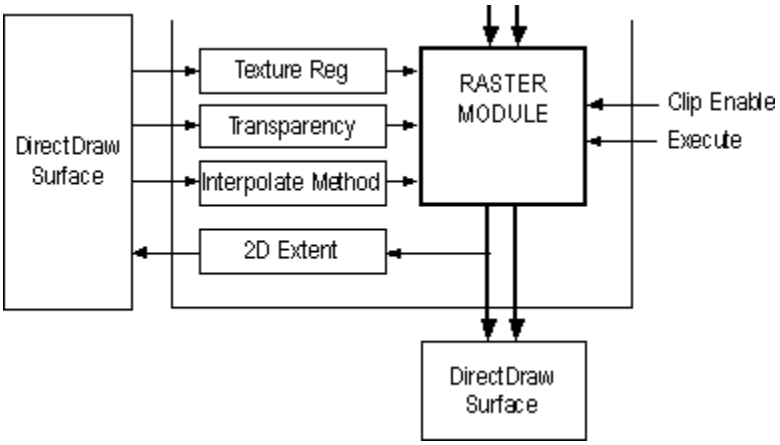
The Direct3D architecture is based on a virtual 3D rendering engine composed of three separate modules:

- The transformation module handles the transformation of geometry by using three 4-by-4 matrices, one each for the view, world, and projection transformations. This module supports arbitrary projection matrices, allowing perspective and orthographic views.
- The lighting module calculates lighting information for geometry, supporting ambient, directional, point, and spotlight light sources.
- The rasterization module uses the output of the geometry and lighting modules to render the scene. The scene description is in an extensible display-list–based format that can support both 2D and 3D primitives.

The following illustration shows how the three modules of the rendering engine interact with the rest of the Direct3D architecture.



The rasterization module interacts with DirectDraw as shown in the following illustration. Direct3D makes use of DirectDraw surfaces as render targets and texture sources.



Each of these modules can be hardware-accelerated or emulated in software. Direct3D can be queried to verify which components are currently running under emulation. When used together, these modules form the Direct3D rendering pipeline.

All three modules are dynamically loadable and can be changed on the fly between rendered frames. This allows new modules to be swapped in for either hardware acceleration or different rendering effects. Direct3D comes with one transformation module but a choice of two lighting and two rasterization modules. This provides greater flexibility in lighting and rendering, allowing the possibility, for example, of rendering more realistic scenes by simply switching the lighting module. An independent vendor could even provide its own special-effects-rasterization modules.

Execute Buffers

Each of the three modules in the rendering engine maintains state information that is set by using the Direct3D API. After all the state information is set, the rendering engine is ready to process display lists, which are known as *execute buffers*. Your application works explicitly with execute buffers only in Immediate Mode; Retained-Mode applications work at a higher level than this.

Execute buffers are fully self-contained, independent packets of information. They contain a vertex list followed by an instruction stream. The instruction stream consists of operation codes, or *opcodes*, and the data that is operated on by those opcodes. Direct3D's opcodes are listed in the D3DOPCODE enumerated type. The D3DINSTRUCTION structure describes instructions in an execute buffer; it contains an opcode, the size of each instruction data unit, and a count of the relevant data units that follow.

The following illustration shows the format of execute buffers.

The instructions define how the vertex list should be lit and rendered. One of the most common instructions is a *triangle list* (D3DOP_TRIANGLE), which is simply a list of triangle primitives that reference vertices in the vertex list. Because all the primitives in the instruction stream reference vertices in the vertex list only, it is easy for the transformation module to reject a whole buffer of primitives if its vertices are outside the viewing frustum.

The hardware determines the size of the execute buffer. You can retrieve this size by calling the IDirect3DDevice::GetCaps method and examining the **dwMaxBufferSize** member of the D3DDEVICEDESC structure. Typically, 64K is a good size for execute buffers when you are using a software driver because this size makes the best use of the secondary cache. When your application can take advantage of hardware acceleration, however, it should use smaller execute buffers to take advantage of the primary cache.

You can disable the lighting module or both the lighting and transformation modules when you are working with execute buffers. This changes the way the vertex list is interpreted, allowing the user to supply pretransformed or prelit vertices only for the rasterization phase of the rendering pipeline. Note that only one vertex type can be used in each execute buffer.

In addition to execute buffers and state changes, Direct3D accepts a third calling mechanism. Either of the transformation or lighting modules can be called directly. This functionality is useful when rasterization is not required, such as when using the transformation module for bounding-box tests.

Transformation Module

The transformation module has four state registers that the user can modify: the viewport, the viewing matrix, the world matrix, and the projection matrix. Whenever one of these parameters is modified, they are combined to form a new transformation matrix, which is also maintained by the transformation module. The transformation matrix defines the rotation and projection of a set of 3D vertices from their model coordinates to the 2D window.

Although the application can set the transformation matrix directly, it is not recommended. A number of matrix classifications take place in the combination phase that allow optimized transformation math to be used, but this is precluded if the application specifies the matrix directly.

A display list supports a number of different vertex types. For rasterization-only hardware, the application should use the D3DTLVERTEX structure, which is a transformed and lit vertex—that is, it contains screen coordinates and colors. If the hardware handles the transformations, the application should use a D3DLVERTEX structure. This structure contains only data and a color that would be filled by software lighting. The D3DHVERTEX structure defines a homogeneous vertex used when the application is supplying model-coordinate data that needs clipping. If the hardware supports lighting, the application simply uses a D3DVERTEX structure, because this type of vertex can be transformed and lit during rendering. The software emulation driver supports all of these vertex types.

There are two types of methods for the transformation module: those that set the state of the transformation module and those that use the transformation module directly to act on a set of vertices. Calling the transformation module directly is useful for testing bounding volumes or for transforming a set of vectors. These operations transform geometry by using the current transformation matrices. They can also perform clipping tests against the current viewing volume. The structure used for all the direct transformation functions is D3DTRANSFORMDATA.

Lighting Module

The lighting module maintains a stack of current lights, the ambient light level, and a material.

When using the lighting module directly, each element of input data to the lighting module (the D3DLIGHTINGELEMENT structure) contains a direction vector and a position (for positional light sources such as point lights and spotlights).

Two lighting models—monochromatic and RGB—are supported. The color fields are always placed after the **D3DLIGHTINGELEMENT** structure in the D3DLIGHTDATA structure.

The monochromatic lighting model (sometimes also called the "ramp" lighting model) uses the gray component of each light to calculate a single shade value. The RGB lighting model produces more realistic results, using the full color content of light sources and materials to calculate the lit colors.

For materials with no specular component, the shade is the diffuse component of the light intensity and ranges from 0 (ambient light only) to 1 (full intensity light). For materials with a specular component, the shade combines both the specular and diffuse components of the light according to the following equation:

$$shade = \frac{3}{4}(diffuse - diffuse \times specular) + specular$$

This shade value is designed to work with precalculated ramps of colors (either in the hardware's color-lookup table or in lookup tables implemented in software). These precalculated ramps are divided into two sections. A ramp of the material's diffuse color takes up the first three-quarters of the precalculated ramp; this ranges from the ambient color to the maximum diffuse color. A ramp ranging from the maximum diffuse color to the maximum specular color of the material takes up the last quarter of the precalculated ramp. For rendering, the shade value should be scaled by the size of the ramp and used as an index to look up the color required.

A packed RGB color is defined as the following:

```
#define RGB_MAKE (red, green, blue) \
    ((red)   << 16) | \
    ((green) << 8)  | \
    (blue))
```

A packed RGBA color is defined as the following:

```
#define RGBA_MAKE(red, green, blue, alpha) \
   (((alpha) << 24) | \
    ((red)   << 16) | \
    ((green) << 8)  | \
    (blue))
```

Colors in Direct3D are defined as follows:

```
typedef unsigned long D3DCOLOR;
```

The type of the light must be one of the members of the D3DLIGHTTYPE enumerated type: D3DLIGHT_DIRECTIONAL, D3DLIGHT_POINT, D3DLIGHT_PARALLELPOINT, D3DLIGHT_SPOT, or D3DLIGHT_GLSPOT. This enumerated type is part of the D3DLIGHT structure. Another member of this structure is a D3DCOLORVALUE structure, which specifies the color of the light. The values given for the red, green, and blue light components typically range from 0 to 1. The value for the ramp lighting model is based on the following equation:

$$shade = 0.30red + 0.59green + 0.11blue$$

Each of the color values can fall outside of the 0 to 1 range, allowing the use of advanced lighting effects (such as dark lights). The direction vectors in the D3DLIGHT structure describe the direction from the model to the light source. This vector should be normalized for directional lights. All vectors should be given in world coordinates; these vectors are transformed into model coordinates by using the current world matrix, allowing the efficient lighting of models without the need to transform the vectors into world coordinates. For point lights and spotlights, the range parameter gives the effective range of the light source. Vertices that are outside this range will not be affected by the light. The intensity of the light is

modified by a quadratic attenuation factor, where *d* is the distance from the vertex being lit to the light, as shown in the following equation:

$$attenuation = attenuation_0 + attenuation_1 \times d + attenuation_2 \times d^2$$

The remaining members of the **D3DLIGHT** structure (**dvTheta** and **dvPhi**) are used for spotlights to define the angles of the umbra and penumbra cones, respectively. The falloff factor (**dvFalloff**) is applied between the umbra and penumbra cones of the spotlight.

There are two types of methods for the lighting module, those that set the state of the lighting module, and those that use the lighting module directly to act on a set of points.

Like the transformation module, the lighting module can also be called directly. The D3DLIGHTDATA structure is used for all the direct lighting functions.

Rasterization Module

The rasterization module handles only execute calls—the calls that render execute buffers. Instructions in the execute buffer set the state for the rasterization module.

Execute buffers are processed first by the transformation module. This module runs through the vertex list, generating transformed vertices by using the state information set up for the transformation module. Clipping can be enabled, generating additional clipping information by using the viewport parameters to clip against. The whole buffer can be rejected here if none of the vertices is in view. Then, the vertices are processed by the lighting module, which adds color to them according to the lighting instructions in the execute buffer. Finally, the rasterization module parses the instruction stream, rendering primitives by using the generated vertex information.

When an application calls the IDirect3DDevice::Execute method, the system determines whether the vertex list needs to be transformed or transformed and lit. After these operations have been completed, the instruction list is parsed and rendered.

The screen coordinates range from (0, 0) for the top left of the device (screen or window) to (*width* – 1, *height* – 1) for the bottom right of the device. The depth values range from 0 at the front of the viewing frustum to 1 at the back. Rasterization is performed so that if two triangles that share two vertices are rendered, no pixel along the line joining the shared vertices is rendered twice. The rasterizer culls backfacing triangles by determining the winding order of the three vertices of the triangle. Only those triangles whose vertices are traversed in a clockwise orientation are rendered.

Colors and Fog

Colors in Direct3D are properties of vertices, textures, materials, faces, lights, and, of course, palettes.

Palette Entries

Your application can use the IDirect3DRM::CreateDeviceFromSurface method to draw to a DirectDraw surface. You must be sure to attach a DirectDraw palette to the primary DirectDraw surface to avoid unexpected colors in Direct3D applications. The Direct3D sample code in this SDK attaches the palette to the primary surface whenever the window receives a WM_ACTIVATE message. If you need to track the changes that Direct3D makes to the palette of an 8-bit DirectDraw surface, you can call the IDirectDrawPalette::GetEntries method.

Your application can use three flags to specify how it will share palette entries with the rest of the system:

D3DPAL_FREE	The renderer may use this entry freely.
D3DPAL_READONLY	The renderer may not set this entry.
D3DPAL_RESERVED	The renderer may not use this entry.

These flags can be specified in the **peFlags** member of the standard Win32 **PALETTEENTRY** structure. (You can also use the members of the D3DRMPALETTEFLAGS enumerated type in the D3DRMPALETTEENTRY structure to specify how to share palette entries.) Your application can use these flags when using either the RGB or monochromatic (ramp) renderer. Although you could supply a read-only palette to the RGB renderer, you will get better results with the ramp renderer.

Fog

Fog is simply the alpha part of the color specified in the **specular** member of the D3DTLVERTEX structure. Another way of thinking about this is that specular color is really RGBF color, where "F" is "fog."

In monochromatic lighting mode, fog works properly only when the fog color is black or when there is no lighting, in which case any fog color has the same effect.

There are three fog modes: linear, exponential, and exponential squared. Only the linear fog mode is supported for DirectX 2.

When you use linear fog, you specify a start and end point for the fog effect. The fog effect begins at the specified starting point and increases linearly until it reaches its maximum density at the specified end point.

The exponential fog modes begin with a barely visible fog effect and increase to the maximum density along an exponential curve. The following is the formula for the exponential fog mode:

$$f = e^{-(density \times z)}$$

In the exponential squared fog mode, the fog effect increases more quickly than in the exponential fog mode. The following is the formula for the exponential squared fog mode:

$$f = e^{-(density \times z)^2}$$

In these formulas, *e* is the base of the natural logarithms; its value is approximately 2.71828. Note that fog can be considered as a measure of visibility—the lower the fog value, the less visible an object is.
For example, if an application used the exponential fog mode and a fog density of 0.5, the fog value at a distance from the camera of 0.8 would be 0.6703, as shown in the following example:

$$f = \frac{1}{2.71828^{(0.5 \times 0.8)}} = \frac{1}{1.4918} = 0.6703$$

States and State Overrides

Direct3D interprets the data in execute buffers according to the current state settings. Applications set up these states before instructing the system to render data. The D3DSTATE structure contains three enumerated types that expose this architecture: D3DTRANSFORMSTATETYPE, which sets the state of the transform module; D3DLIGHTSTATETYPE, for the lighting module; and D3DRENDERSTATETYPE, for the rasterization module.

Each state includes a Boolean value that is essentially a read-only flag. If this flag is set to TRUE, no further state changes are allowed.

Applications can override the read-only state of a module by using the D3DSTATE_OVERRIDE macro. This mechanism allows an application to reuse an execute buffer, changing its behavior by changing the system's state. Direct3D Retained Mode uses state overrides to accomplish some tasks that otherwise would require completely rebuilding an execute buffer. For example, the Retained-Mode API uses state overrides to replace the material of a mesh with the material of a frame.

An application might use the **D3DSTATE_OVERRIDE** macro to lock and unlock the Gouraud shade mode, as shown in the following example. (The shade-mode render state is defined by the D3DRENDERSTATE_SHADEMODE member of the **D3DRENDERSTATETYPE** enumerated type.)

```
OP_STATE_RENDER(2, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_SHADEMODE, D3DSHADE_GOURAUD, lpBuffer);
    STATE_DATA(D3DSTATE_OVERRIDE(D3DRENDERSTATE_SHADEMODE), TRUE, lpBuffer);
```

The OP_STATE_RENDER macro implicitly uses the D3DOP_STATERENDER opcode, one of the members of the **D3DOPCODE** enumerated type. D3DSHADE_GOURAUD is one of the members of the **D3DSHADEMODE** enumerated type.

After executing the execute buffer, the application could use the **D3DSTATE_OVERRIDE** macro again, to allow the shade mode to be changed:

```
    STATE_DATA(D3DSTATE_OVERRIDE(D3DRENDERSTATE_SHADEMODE), FALSE,
lpBuffer);
```

The OP_STATE_RENDER and STATE_DATA macros are defined in the D3dmacs.h header file in the Misc directory of the DirectX 2 SDK sample code; these macros are also described in Setting the Immediate-Mode Render State.

Direct3D File Format

The Direct3D file format stores meshes, textures, animation sets, and user-definable objects, facilitating the exchange of 3D information between applications. Support for animation sets allows predefined paths to be stored for playback in real time. Instancing and hierarchies are also supported, allowing multiple references to a single data object (such as a mesh) while storing the data for the object only once per file.

Direct3D files have a .X file name extension. This DirectX™ 2 Software Development Kit (SDK) includes conversion tools (Conv3ds.exe and Convxof.exe) that allow you to convert files from 3DS files generated by Autodesk 3D Studio® and from XOF files that were generated for earlier versions of Direct3D.

The Direct3D file format is used natively by the Direct3D Retained-Mode API, providing support for loading predefined objects into an application and for writing mesh information constructed by the application in real time.

A Technical Foundation for 3D Programming

The following sections describe some of the technical concepts you need to understand before you write programs that incorporate 3D graphics. In these sections, you will find a general discussion of coordinate systems and transformations. This is not a discussion of broad architectural details, such as setting up models, lights, and viewing parameters. For more information about these topics, see Introduction to Direct3D Retained-Mode Objects.

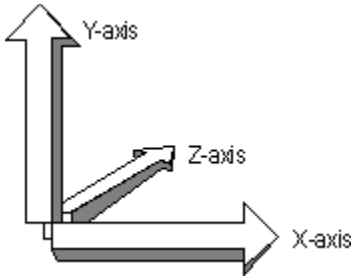
If you are already experienced in producing 3D graphics, simply scan the following sections for information that is unique to Direct3D Retained Mode.

3D Coordinate Systems

There are two varieties of Cartesian coordinate systems in 3D graphics: left-handed and right-handed. In both coordinate systems, the positive x-axis points to the right and the positive y-axis points up. You can remember which direction the positive z-axis points by pointing the fingers of either your left or right hand in the positive x direction and curling them into the positive y direction. The direction your thumb points, either toward or away from you, is the direction the positive z-axis points for that coordinate system.

Direct3D's Coordinate System

Direct3D uses the left-handed coordinate system. This means the positive z-axis points away from the viewer, as shown in the following illustration:



In a left-handed coordinate system, rotations occur clockwise around any axis that is pointed at the viewer.

If you need to work in a right-handed coordinate system—for example, if you are porting an application that relies on right-handedness—you can do so by making two simple changes to the data passed to Direct3D.

- Flip the order of triangle vertices so that the system traverses them clockwise from the front. In other words, if the vertices are v0, v1, v2, pass them to Direct3D as v0, v2, v1.
- Scale the projection matrix by -1 in the z direction. To do this, flip the signs of the _13, _23, _33, and _43 members of the D3DMATRIX structure.

U- and V-Coordinates

Direct3D also uses *texture coordinates*. These coordinates (u and v) are used when mapping textures onto an object. The v-vector describes the direction or orientation of the texture and lies along the z-axis. The u-vector (or the *up* vector) typically lies along the y-axis, with its origin at [0,0,0]. For more information about u- and v-coordinates, see Direct3DRMWrap.

3D Transformations

In programs that work with 3D graphics, you can use geometrical transformations to:

- Express the location of an object relative to another object.
- Rotate, shear, and size objects.
- Change viewing positions, directions, and perspective.

You can transform any point into another point by using a 4-by-4 matrix. In the following example, a matrix is used to reinterpret the point (x, y, z), producing the new point (x', y', z'):

$$[x'y'z'1] = [x\,y\,z\,1]\begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

You perform the following operations on (x, y, z) and the matrix to produce the point (x', y', z'):

$$x' = (M_{11} \times x) + (M_{21} \times y) + (M_{31} \times z) + (M_{41} \times 1)$$
$$y' = (M_{12} \times x) + (M_{22} \times y) + (M_{32} \times z) + (M_{42} \times 1)$$
$$z' = (M_{13} \times x) + (M_{23} \times y) + (M_{33} \times z) + (M_{43} \times 1)$$

The most common transformations are translation, rotation, and scaling. You can combine the matrices that produce these effects into a single matrix to calculate several transformations at once. For example, you can build a single matrix to translate and rotate a series of points.

Matrices are specified in row order. For example, the following matrix could be represented by an array:

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & t & 0 \\ 0 & 0 & s & v \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The array for this matrix would look like the following:

```
D3DMATRIX scale = {
    D3DVAL(s),    0,            0,            0,
    0,            D3DVAL(s),    D3DVAL(t),    0,
    0,            0,            D3DVAL(s),    D3DVAL(v),
    0,            0,            0,            D3DVAL(1)
};
```

Translation

The following transformation translates the point (x, y, z) to a new point (x', y', z'):

$$[x'y'z'1] = [x\,y\,z\,1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Rotation

The transformations described in this section are for left-handed coordinate systems, and so may be different from transformation matrices you have seen elsewhere.

The following transformation rotates the point (x, y, z) around the x-axis, producing a new point (x', y', z'):

$$[x'y'z'1] = [x\,y\,z\,1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following transformation rotates the point around the y-axis:

$$[x'y'z'1] = [x\,y\,z\,1]\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following transformation rotates the point around the z-axis:

$$[x'y'z'1] = [x\,y\,z\,1]\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that in these example matrices, the Greek letter theta stands for the angle of rotation, specified in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Scaling

The following transformation scales the point (x, y, z) by arbitrary values in the x-, y-, and z-directions to a new point (x', y', z'):

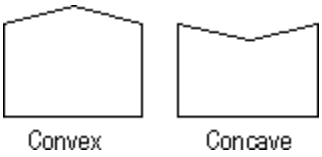
$$[x'y'z'1] = [x\,y\,z\,1]\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Polygons

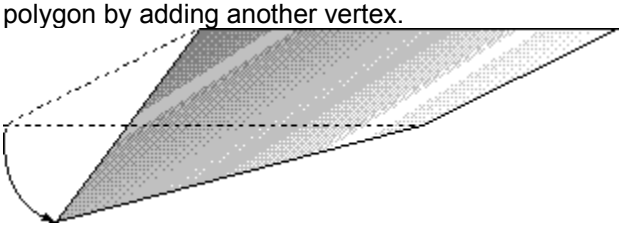
Three-dimensional objects in Direct3D are made up of meshes. A mesh is a set of faces, each of which is described by a simple polygon. The fundamental polygon type is the triangle. Although Retained-Mode applications can specify polygons with more than three vertices, the system translates these into triangles before the objects are rendered. Immediate-Mode applications must use triangles.

Geometry Requirements

Triangles are the preferred polygon type because they are always convex and they are always planar, two conditions that are required of polygons by the renderer. A polygon is convex if a line drawn between any two points of the polygon is also inside the polygon.

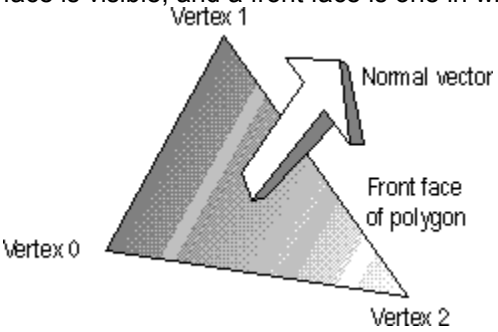


The three vertices of a triangle always describe a plane, but it is easy to accidentally create a non-planar polygon by adding another vertex.

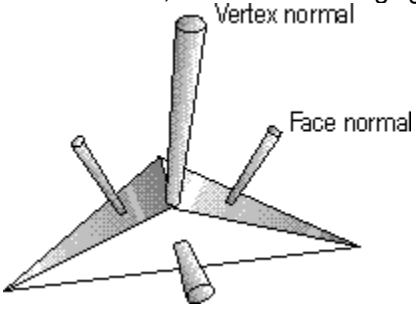


Face and Vertex Normals

Each face in a mesh has a perpendicular face normal whose direction is determined by the order in which the vertices are defined and by whether the coordinate system is right- or left-handed. If the normal vector of a face is oriented toward the viewer, that side of the face is its front. In Direct3D, only the front side of a face is visible, and a front face is one in which vertices are defined in clockwise order.



Direct3D applications do not need to specify face normals; the system calculates them automatically when they are needed. The system uses face normals in the flat shade mode. For Phong and Gouraud shade modes, and for controlling lighting and texturing effects, the system uses vertex normals.



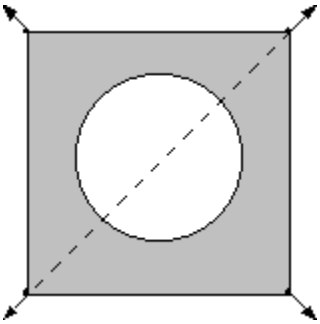
Shade Modes

In the flat shade mode, the system duplicates the color of one vertex across all the other faces of the primitive. In the Gouraud and Phong shade modes, vertex normals are used to give a smooth look to a polygonal object. In Gouraud shading, the color and intensity of adjacent vertices is interpolated across

the space that separates them. In Phong shading, the system calculates the appropriate shade value for each pixel on a face.

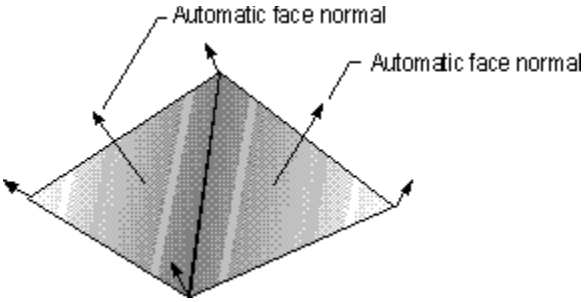
Note Phong shading is not supported for DirectX 2.

Most applications use Gouraud shading because it allows objects to appear smooth and is computationally efficient. Gouraud shading can miss details that Phong shading will not, however. For example, Gouraud and Phong shading would produce very different results in the case shown by the following illustration, in which a spotlight is completely contained within a face.

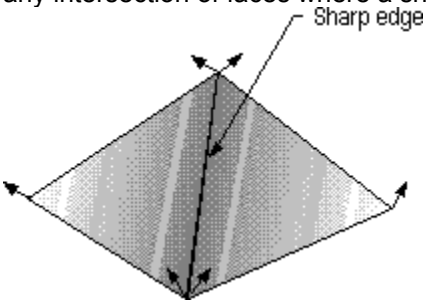


In this case, the Phong shade mode would calculate the value for each pixel and display the spotlight. The Gouraud shade mode, which interpolates between vertices, would miss the spotlight altogether; the face would be rendered as though the spotlight did not exist.

In the flat shade mode, the following pyramid would be displayed with a sharp edge between adjoining faces; the system would generate automatic face normals. In the Gouraud or Phong shade modes, however, shading values would be interpolated across the edge, and the final appearance would be of a curved surface.



If you want to use the Gouraud or Phong shade mode to display curved surfaces and you also want to include some objects with sharp edges, your application would need to duplicate the vertex normals at any intersection of faces where a sharp edge was required, as shown in the following illustration.



In addition to allowing a single object to have both curved and flat surfaces, the Gouraud shade mode lights flat surfaces more realistically than the flat shade mode. A face in the flat shade mode is a uniform color, but Gouraud shading allows light to fall across a face correctly; this effect is particularly obvious if there is a nearby point source. Gouraud shading is the preferred shade mode for most Direct3D applications.

Triangle Interpolants

The system interpolates the characteristics of a triangle's vertices across the triangle when it renders a face. The following are the triangle interpolants:

- Color
- Specular
- Fog
- Alpha

All of the triangle interpolants are modified by the current shade mode:

Flat	No interpolation is done. Instead, the color of the first vertex in the triangle is applied across the entire face.
Gouraud	Linear interpolation is performed between all three vertices.
Phong	Vertex parameters are reevaluated for each pixel in the face, using the current lighting. The Phong shade mode is not supported for DirectX 2.

The color and specular interpolants are treated differently, depending on the color model. In the RGB color model (D3DCOLOR_RGB), the system uses the red, green, and blue color components in the interpolation. In the monochromatic model (D3DCOLOR_MONO), the system uses only the blue component of the vertex color.

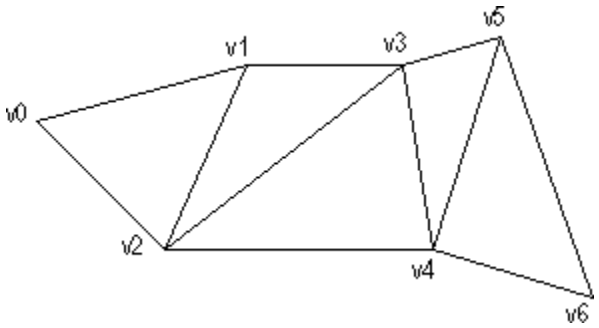
For example, if the red component of the color of vertex 1 were 0.8 and the red component of vertex 2 were 0.4, in the Gouraud shade mode and RGB color model the system would use interpolation to assign a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

The alpha component of a color is treated as a separate interpolant because device drivers can implement transparency in two different ways: by using texture blending or by using stippling.

An application can use the **dwShadeCaps** member of the D3DPRIMCAPS structure to determine what forms of interpolation the current device driver supports.

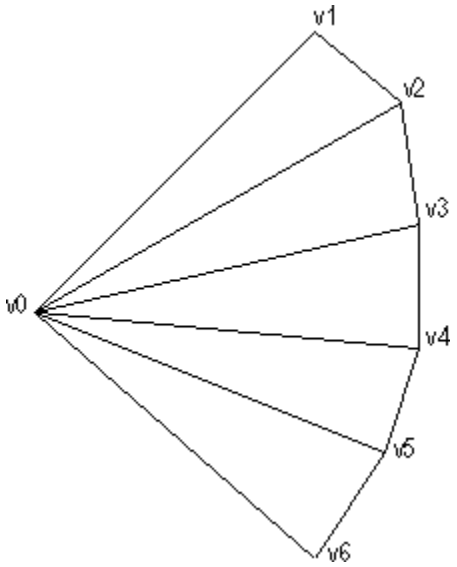
Triangle Strips and Fans

You can use triangle strips and triangle fans to specify an entire surface without having to provide all three vertices for each of the triangles. For example, only seven vertices are required to define the following triangle strip.



The system uses vertices v0, v1, and v2 to draw the first triangle, v1, v3, and v2 to draw the second triangle, v3, v4, and v2 to draw the third, and so on. Notice that the vertices of the second triangle are out of order; this is required to make sure that all of the triangles are drawn in a clockwise orientation.

A triangle fan is similar to a triangle strip, except that all of the triangles share one vertex.



The system uses vertices v0, v1, and v2 to draw the first triangle, v0, v2, and v3 to draw the second triangle, and so on.

You can use the **wFlags** member of the D3DTRIANGLE structure to specify the flags that build triangle strips and fans.

Vectors and Quaternions

Vectors are used throughout Direct3D to describe position and orientation. Each vertex in a primitive is described by a vector that gives its position, a normal vector that gives its orientation, texture coordinates, and a color. (In Retained Mode, the D3DRMVERTEX structure contains these values.)

Quaternions add a fourth element to the [*x, y, z*] values that define a vector. They are an alternative to the matrix methods that are typically used for 3D rotations. Direct3D's Retained Mode includes some functions that help you work with quaternions. For example, the D3DRMQuaternionFromRotation function adds a rotation value to a vector that defines an axis of rotation and returns the result in a quaternion defined by a D3DRMQUATERNION structure.

Retained-Mode applications can use the following functions to simplify the task of working with vectors and quaternions:

D3DRMQuaternionFromRotation
D3DRMQuaternionMultiply
D3DRMQuaternionSlerp
D3DRMVectorAdd
D3DRMVectorCrossProduct
D3DRMVectorDotProduct
D3DRMVectorModulus
D3DRMVectorNormalize
D3DRMVectorRandom
D3DRMVectorReflect
D3DRMVectorRotate
D3DRMVectorScale
D3DRMVectorSubtract

Floating-point Precision

Direct3D, like the rest of the DirectX architecture, uses a floating-point precision of 53 bits. If your application needs to change this precision, it must change it back to 53 when the calculations are finished. Otherwise, system components that depend on the default value will stop working.

Performance Optimization

Every developer who creates real-time applications that use 3D graphics is concerned about performance optimization. This section provides you with guidelines about getting the best performance from your code.

You can use the guidelines in the following sections for any Direct3D application:

- Clip Tests on Execution
- Batching Primitives
- Texture Size
- Triangle Flags

Direct3D applications can use either the ramp driver (for the monochromatic color model) or the RGB driver. The performance notes in the following sections apply to the ramp driver:

- Ramp Performance Tips
- Ramp Textures
- Z-Buffers
- Copy Mode

Clip Tests on Execution

Your application can use the IDirect3DDevice::Execute method to render primitives with or without automatic clipping. Using this method without clipping is always faster than setting the clipping flags because clipping tests during either the transformation or rasterization stages slow the process. If your application does not use automatic clipping, however, it must ensure that all of the rendering data is wholly within the viewing frustum. The best way to ensure this is to use simple bounding volumes for the models and transform these first. You can use the results of this first transformation to decide whether to wholly reject the data because all the data is outside the frustum, whether to use the no-clipping version of the **IDirect3DDevice::Execute** method because all the data is within the frustum, or whether to use the clipping flags because the data is partially within the frustum. In Immediate Mode it is possible to set up this sort of functionality within one execute buffer by using the flags in the D3DSTATUS structure and the D3DOP_BRANCHFORWARD member of the **D3DOPCODE** enumerated type to skip geometry when a bounding volume is outside the frustum. Direct3D's Retained Mode automatically uses these features to speed up its use of execute buffers.

Batching Primitives

To get the best rendering performance during execution, you should try to work with primitives in batches and keep the number of render-state changes as low as possible. For example, if you have an object with two textures, group the triangles that use the first texture and follow them with the necessary render state to change the texture, then group all the triangles that use the second texture. The simplest hardware support for Direct3D is called with batches of render states and batches of primitives through the hardware-abstraction layer (HAL). The more effectively the instructions are batched, the fewer HAL calls are performed during execution.

Texture Size

Texture-mapping performance is heavily dependent on the speed of memory. There are a number of ways to maximize the cache performance of your application's textures.

- Keep the textures small; the smaller the textures are, the better chance they have of being maintained in the main CPU's secondary cache.
- Do not change the textures on a per-primitive basis. Try to keep polygons grouped in order of the textures they use.
- Use square textures whenever possible. Textures whose dimensions are 256 by 256 are the fastest. If your application uses four 128-by-128 textures, for example, try to ensure that they use the same palette and place them all into one 256-by-256 texture. This technique also reduces the amount of texture swapping. Of course, you should not use 256-by-256 textures unless your application requires that much texturing because, as already mentioned, textures should be kept as small as possible.

Triangle Flags

The **wFlags** member of the D3DTRIANGLE structure includes flags that allow the system to reuse vertices when building triangle strips and fans. Effective use of these flags allows some hardware to run much faster than it would otherwise.

Applications can use these flags in two ways as acceleration hints to the driver:

D3DTRIFLAG_STARTFLAT(*len*)

If the current triangle is culled, the driver can also cull the number of subsequent triangles given by *len* in the strip or fan.

D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN

The driver needs to reload only one new vertex from the triangle and it can reuse the other two vertices from the last triangle that was rendered.

The best possible performance occurs when an application uses both the D3DTRIFLAG_STARTFLAT flag and the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags.

Because some drivers might not check the D3DTRIFLAG_STARTFLAT flag, applications must be careful when using it. An application using a driver that doesn't check this flag might not render polygons that should have been rendered.

Applications must use the D3DTRIFLAG_START flag before using the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags. D3DTRIFLAG_START causes the driver to reload all three vertices. All triangles following the D3DTRIFLAG_START flag can use the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags indefinitely, providing the triangles share edges.

The debugging version of this SDK validates the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags.

For more information, see Triangle Strips and Fans.

Ramp Performance Tips

Applications should use the following techniques to achieve the best possible performance when using the monochromatic (ramp) driver:

- Share the same palette among all textures.
- Keep the number of colors in the palette as low as possible—64 or fewer is best.
- Keep the ramp size in materials at 16 or less.
- Make all materials the same (except the texture handle)—allow the textures to specify the coloring. For example, make all the materials white and keep their specular power the same. Many applications do not need more than two materials in a scene: one with a specular power for shiny objects, and one without for matte objects.
- Keep textures as small as possible.
- Fit multiple small textures into a single texture that is 256-by-256 pixels.
- Render small triangles by using the Gouraud shade mode, and render large triangles by using the flat shade mode.

Developers who must use more than one palette can optimize their code by using one palette as a master palette and ensuring that the other palettes contain a subset of the colors in the master palette.

Ramp Textures

Applications that use the ramp driver should be conservative with the number of texture colors they require. Each color used in a monochromatic texture requires its own lookup table during rendering. If your application uses hundreds of colors in a scene during rendering, the system must use hundreds of lookup tables, which do not cache well. Also, try to share palettes between textures whenever possible. Ideally, all of your application's textures will fit into one palette, even when you are using a ramp driver with depths greater than 8-bit color.

Z-Buffers

Applications that use the ramp driver can increase performance when using z-buffering and texturing by ensuring that scenes are rendered from front to back. Textured z-buffered primitives are pretested against the z-buffer on a scanline basis. If a scanline is hidden by a previously rendered polygon, the system rejects it quickly and efficiently. Z-buffering can improve performance, but the technique is most useful when a scene includes a great deal of *overdraw*. Overdraw is the average number of times a screen pixel is written to. Overdraw is difficult to calculate exactly, but you can often make a close approximation. If the overdraw averages less than 2, you can achieve the best performance by turning z-buffering off.

You can also improve the performance of your application by z-testing primitives; that is, by testing a given list of primitives against the z-buffer. This allows for fast bounding-box rejection of occluded geometry.

The Retained-Mode API can automatically order its scenes from front to back to facilitate z-buffer optimization. Retained Mode also z-tests primitives for all meshes that contain more than a few hundred triangles.

You can use the fill-rate test in the D3dtest.exe application that is provided with this SDK to demonstrate overdraw performance for a given driver. (The fill-rate test draws four tunnels from front to back or back to front, depending on the setting you choose.)

On faster personal computers, software rendering to system memory is often faster than rendering to video memory, although it has the disadvantage of not being able to use double buffering or hardware-accelerated clear operations. If your application can render to either system or video memory, and if you include a routine that tests which is faster, you can take advantage of the best approach on the current system. The Direct3D sample code in this SDK demonstrates this strategy. It is necessary to implement both methods because there is no other way to test the speed. Speeds can vary enormously from computer to computer, depending on the main-memory architecture and the type of graphics adapter being used. Although you can use D3dtest.exe to test the speed of system memory against video memory, it cannot predict the performance of your user's personal computer.

You can run all of the Direct3D samples in system memory by using the "-systemmemory" command-line option. This is also useful when developing code because it allows your application to fail in a way that stops the renderer without stopping your system—DirectDraw does not take the WIN16 lock for system-memory surfaces. (The WIN16 lock serializes access to GDI and USER, shutting down Windows for the interval between calls to the IDirectDrawSurface::Lock and IDirectDrawSurface::Unlock methods, as well as between calls to the IDirectDrawSurface::GetDC and IDirectDrawSurface::ReleaseDC methods.)

Copy Mode

Applications that use the ramp driver can sometimes improve performance by using the D3DTBLEND_COPY texture-blending mode from the D3DTEXTUREBLEND enumerated type.

To use copy mode, your application's textures must use the same pixel format as the primary surface and also must use the same palette as the primary surface. Copy mode does no lighting and simply copies texture pixels to the screen. This is often a good technique for prelit textured scenes.

If your application uses the monochromatic model with 8-bit color and no lighting, performance can improve if you use copy mode. If your application uses 16-bit color, however, copy mode is not quite as fast as using modulated textures; for 16-bit color, textures are twice the size as in the 8-bit case, and the extra burden on the cache makes performance slightly worse than using an 8-bit lit texture. Again, you can use D3dtest.exe to verify system performance in this case.

About Retained Mode

This section describes Direct3D's Retained Mode, Microsoft's solution for real-time 3D graphics on the personal computer. If you need to create a 3D environment and manipulate it in real time, you should use Direct3D's Retained-Mode API.

Direct3D is integrated tightly with DirectDraw. A DirectDraw object encapsulates both the DirectDraw and Direct3D states—your application can use the IDirectDraw::QueryInterface method to retrieve an IDirect3D interface to a DirectDraw object. For more information about this integration, see Direct3D Driver Interface.

If you have written code that uses 3D graphics before, many of the concepts underlying Retained Mode will be familiar to you. If, however, you are new to 3D programming, you should pay close attention to Introduction to Direct3D Retained-Mode Objects, and you should read A Technical Foundation for 3D Programming. Whether you are new to 3D programming or just beginning, you should look carefully at the sample code included with this SDK; it illustrates how to put Retained Mode to work in real-world applications.

This section is an introduction to 3D programming. It describes Microsoft's 3D-graphics solutions and some of the technical background you need to manipulate points in three dimensions. It is not an introduction to programming with Direct3D's Retained Mode; for this information, see Direct3D Retained-Mode Tutorial.

Introduction to Direct3D Retained-Mode Objects

All access to Direct3D Retained Mode is through a small set of objects. The following table lists these objects and a brief description of each:

Object	Description
Direct3DRMAnimation	This object defines how a transformation will be modified, often in reference to a Direct3DRMFrame object; therefore, you can use it to animate the position, orientation, and scaling of Direct3DRMVisual, Direct3DRMLight, and Direct3DRMViewport objects.
Direct3DRMAnimationSet	This object allows Direct3DRMAnimation objects to be grouped together.
Direct3DRMDevice	This object represents the visual display destination for the renderer.
Direct3DRMFace	This object represents a single polygon in a mesh.
Direct3DRMFrame	This object positions objects within a scene and defines the positions and orientations of visual objects.
Direct3DRMLight	This object defines one of five types of lights that are used to illuminate the visual objects in a scene.
Direct3DRMMaterial	This object defines how a surface reflects light.
Direct3DRMMesh	This object consists of a set of polygonal faces. You can use this object to manipulate groups of faces and vertices.
Direct3DRMMeshBuilder	This object allows you to work with individual vertices and faces in a mesh.
Direct3DRMObject	This object is a base class used by all other Direct3D Retained-Mode objects; it has characteristics that are common to all objects.
Direct3DRMPickedArray	This object identifies a visual object that corresponds to a given 2D point.
Direct3DRMShadow	This object defines a shadow.
Direct3DRMTexture	This object is a rectangular array of colored pixels.
Direct3DRMUserVisual	This object is defined by an application to provide functionality not otherwise available in the system.
Direct3DRMViewport	This object defines how the 3D scene is rendered into a 2D window.
Direct3DRMVisual	This object is anything that can be rendered in a scene. Visual objects

	need not be visible; for example, a frame can be added as a visual object.
Direct3DRMWrap	This object calculates texture coordinates for a face or mesh.

Many objects can be grouped into arrays, called *array objects*. Array objects make it simpler to apply operations to the entire group. The COM interfaces that allow you to work with array objects contain the **GetElement** and **GetSize** methods. These methods retrieve a pointer to an element in the array and the size of the array, respectively. For more information about array interfaces, see Introduction to Array Interfaces.

Objects and Interfaces

Calling the *IObjectName*::**QueryInterface** method retrieves a valid interface pointer only if the object supports that interface; therefore, you could call the **IDirect3DRMDevice::QueryInterface** method to retrieve the **IDirect3DRMWinDevice** interface, but not to retrieve the **IDirect3DRMVisual** interface.

Object name	Supported interfaces
Direct3DRMAnimation	IDirect3DRMAnimation
Direct3DRMAnimationSet	IDirect3DRMAnimationSet
Direct3DRMDevice	IDirect3DRMDevice, IDirect3DRMWinDevice
Direct3DRMFace	IDirect3DRMFace
Direct3DRMFrame	IDirect3DRMFrame, **IDirect3DRMVisual**
Direct3DRMLight	IDirect3DRMLight
Direct3DRMMaterial	IDirect3DRMMaterial
Direct3DRMMesh	IDirect3DRMMesh, **IDirect3DRMVisual**
Direct3DRMMeshBuilder	IDirect3DRMMeshBuilder, **IDirect3DRMVisual**
Direct3DRMShadow	IDirect3DRMShadow, **IDirect3DRMVisual**
Direct3DRMTexture	IDirect3DRMTexture, **IDirect3DRMVisual**
Direct3DRMUserVisual	IDirect3DRMUserVisual, **IDirect3DRMVisual**
Direct3DRMViewport	IDirect3DRMViewport
Direct3DRMWrap	IDirect3DRMWrap

The following code sample illustrates how to create two interfaces to a single Direct3DRMDevice object. The IDirect3DRM::CreateObject method creates an uninitialized Direct3DRMDevice object. The IDirect3DRMDevice::InitFromClipper method initializes the object. The call to the IDirect3DRMDevice::QueryInterface method creates a second interface to the Direct3DRMDevice object—an IDirect3DRMWinDevice interface the application will use to handle WM_PAINT and WM_ACTIVATE messages.

```
d3drmapi->CreateObject(CLSID_CDirect3DRMDevice, NULL,
    IID_IDirect3DRMDevice,(LPVOID FAR*)&dev1);
dev1->InitFromClipper(lpDDClipper, IID_IDirect3DRMDevice,
    r.right, r.bottom);
dev1->QueryInterface(IID_IDirect3DRMWinDevice, (LPVOID*) &dev2);
```

To determine if two interfaces refer to the same object, call the **QueryInterface** method of each interface and compare the values of the pointers they return. If the pointer values are the same, the interfaces refer to the same object.

All Direct3D Retained-Mode objects support the IDirect3DRMObject and IUnknown interfaces in addition to the interfaces in the preceding list. Array objects are not derived from **IDirect3DRMObject**, however. Array objects have no class identifiers (CLSIDs) because they are not needed. Applications cannot create array objects in a call to the IDirect3DRM::CreateObject method; instead, they should use the creation methods listed below for each interface:

Array interface	Creation method
IDirect3DRMDeviceArray	IDirect3DRM::GetDevices
IDirect3DRMFaceArray	IDirect3DRMMeshBuilder::GetFaces
IDirect3DRMFrameArray	IDirect3DRMPickedArray::GetPick or IDirect3DRMFrame::GetChildren
IDirect3DRMLightArray	IDirect3DRMFrame::GetLights
IDirect3DRMPickedArray	IDirect3DRMViewport::Pick
IDirect3DRMViewportArray	IDirect3DRM::CreateFrame
IDirect3DRMVisualArray	IDirect3DRMFrame::GetVisuals

Objects and Reference Counting

Whenever an object is created, its reference count is increased. Each time an application creates a child of an object or a method returns a pointer to an object, the system increases the reference count for that object. The object is not deleted until its reference count reaches zero.

Applications should need to keep track of the reference count for a single object only: the root of the scene. The system keeps track of the other reference counts automatically. Applications should be able to simply release the scene, the viewport, and the device when they clean up before exiting. (When your application releases the viewport, the system automatically takes care of the camera's references.) An application could theoretically release a viewport without releasing the device, such as if it needed to add a new viewport to the device, but whenever an application releases a device, it should release the viewport as well.

The reference count of a frame object increases whenever a child frame or visual object is added to the frame. When you use the IDirect3DRMFrame::AddChild method to move a child from one parent to another, the system handles the reference counting automatically.

After your application loads a visual object into a scene, the scene handles the reference counting for the visual object. The application no longer needs the visual object and can release it.

Creating and applying a wrap does not increase the reference count of any objects, because wrapping is really just a convenient method of calculating texture coordinates.

Direct3DRMAnimation and Direct3DRMAnimationSet

An animation in Retained Mode is defined by a set of *keys*. A key is a time value associated with a scaling operation, an orientation, or a position. A Direct3DRMAnimation object defines how a transformation is modified according to the time value. The animation can be set to operate on a Direct3DRMFrame object, so it could be used to animate the position, orientation, and scaling of Direct3DRMVisual, Direct3DRMLight, and Direct3DRMViewport objects.

TheIDirect3DRMAnimation::AddPositionKey, IDirect3DRMAnimation::AddRotateKey, and IDirect3DRMAnimation::AddScaleKey methods each specify a time value whose units are arbitrary. If an application adds a position key with a time value of 99, for example, a new position key with a time value of 49 would occur exactly halfway between the (zero-based) beginning of the animation and the first position key.

The animation is driven by calling the IDirect3DRMAnimation::SetTime method. This sets the visual object's transformation to the interpolated position, orientation, and scale of the nearby keys in the animation. As with the methods that add animation keys, the time value for **IDirect3DRMAnimation::SetTime** is an arbitrary value, based on the positions of keys the application has already added.

A Direct3DRMAnimationSet object allows Direct3DRMAnimation objects to be grouped together. This allows all the animations in an animation set to share the same time parameter, simplifying the playback of complex articulated animation sequences. An application can add an animation to an animation set by using the IDirect3DRMAnimationSet::AddAnimation method, and it can remove one by using the IDirect3DRMAnimationSet::DeleteAnimation method. Animation sets are driven by calling the IDirect3DRMAnimationSet::SetTime method.

For related information, see the IDirect3DRMAnimation and IDirect3DRMAnimationSet interfaces.

Direct3DRMDevice and Direct3DRMDeviceArray

All forms of rendered output must be associated with an output device. The device object represents the visual display destination for the renderer.

The renderer's behavior depends on the type of output device that is specified. You can define multiple viewports on a device, allowing different aspects of the scene to be viewed simultaneously. You can also specify any number of devices, allowing multiple destination devices for the same scene.

Retained Mode supports devices that render directly to the screen, to windows, or into application memory.

For related information, see the IDirect3DRMDevice interface.

Quality

The device allows the scene and its component parts to be rendered with various degrees of realism. Each mesh can have its own quality, but the maximum quality available for a mesh is that of the device.

An application can change the rendering quality of a device by using the IDirect3DRMDevice::SetQuality and IDirect3DRMMeshBuilder::SetQuality methods. To retrieve the rendering quality for a device, it can use the IDirect3DRMDevice::GetQuality and IDirect3DRMMeshBuilder::GetQuality methods.

Color Models

Retained Mode supports two color models: an RGB model and a monochromatic (or ramp) model. To retrieve the color model, an application can use the IDirect3DRMDevice::GetColorModel method.

The RGB model treats color as a combination of red, green, and blue light, and it supports multiple light sources that can be colored. There is no limit to the number of colors in the scene. You can use this model with 8-, 16-, 24-, and 32-bit displays. If the display depth is less than 24 bits, the limited color resolution can produce banding artifacts; you can avoid these artifacts by using optional dithering.

The monochromatic model also supports multiple light sources, but their color content is ignored. Each source is set to a gray intensity. RGB colors at a vertex are interpreted as brightness levels, which (in Gouraud shading) are interpolated across a face between vertices with different brightnesses. The number of differently colored objects in the scene is limited; after all the system's free palette entries are used up, the system's internal palette manager finds colors that already exist in the palette and that most closely match the intended colors. Like the RGB model, you can use this model with 8-, 16-, 24-, and 32-bit displays. (The monochromatic model supports only 8-bit textures, however.) The advantage of the monochromatic model over the RGB model is simply performance.

It is not possible to change the color model of a Direct3D device. Your application should use the IDirect3D::EnumDevices or IDirect3D::FindDevice method to identify a driver that supports the required color model, then specify this driver in one of the device-creation methods.

Window Management

For correct operation, applications must inform Retained Mode when WM_MOVE, WM_PAINT, and WM_ACTIVATE messages are received from the operating system by using the IDirect3DRMWinDevice::HandlePaint and IDirect3DRMWinDevice::HandleActivate methods.

For related information, see IDirect3DRMWinDevice.

Direct3DRMFace and Direct3DRMFaceArray

A face represents a single polygon in a mesh. An application can set the color, texture, and material of the face by using the IDirect3DRMFace::SetColor, IDirect3DRMFace::SetColorRGB, IDirect3DRMFace::SetTexture, and IDirect3DRMFace::SetMaterial methods.

Faces are constructed from vertices by using the IDirect3DRMFace::AddVertex and IDirect3DRMFace::AddVertexAndNormalIndexed methods. An application can read the vertices of a face by using the IDirect3DRMFace::GetVertices and IDirect3DRMFace::GetVertex methods.

For related information, see IDirect3DRMFace.

Direct3DRMFrame and Direct3DRMFrameArray

The term *frame* is derived from an object's physical frame of reference. The frame's role in Retained Mode is similar to a window's role in a windowing system. Objects can be placed in a scene by stating their spatial relationship to a relevant reference frame; they are not simply placed in world space. A frame is used to position objects in a scene, and visuals take their positions and orientation from frames.

A *scene* in Retained Mode is defined by a frame that has no parent frame; that is, a frame at the top of the hierarchy of frames. This frame is also sometimes called a *root frame* or *master frame*. The scene defines the frame of reference for all of the other objects. You can create a scene by calling the IDirect3DRM::CreateFrame function and specifying NULL for the first parameter.

You should be comfortable with Direct3D's left-handed coordinate system before working with frames. For more information about coordinate systems, see 3D Coordinate Systems.

Hierarchies

The frames in a scene are arranged in a tree structure. Frames can have a parent frame and child frames. Remember, a frame that has no parent frame defines a scene.

Child frames have positions and orientations relative to their parent frames. If the parent frame moves, the child frames also move.

An application can set the position and orientation of a frame relative to any other frame in the scene, including the root frame if it needs to set an absolute position. You can also remove frames from one parent frame and add them to another at any time by using the IDirect3DRMFrame::AddChild method. To remove a child frame entirely, use the IDirect3DRMFrame::DeleteChild method. To retrieve a frame's child and parent frames, use the IDirect3DRMFrame::GetChildren and IDirect3DRMFrame::GetParent methods.

You can add frames as visuals to other frames, allowing you to use a given hierarchy many times throughout a scene. The new hierarchies are referred to as *instances*. Be careful to avoid instancing a parent frame into its children, because that will degrade performance. Retained Mode does no run-time checking for cyclic hierarchies. You cannot create a cyclic hierarchy by using the methods of the IDirect3DRMFrame interface; instead, this is possible only when you add a frame as a visual.

Transformations

You can also think of the position and orientation of a frame relative to its parent frame as a linear transformation that takes vectors defined relative to the child frame and changes them to equivalent vectors defined relative to the parent.

Transformations can be represented by 4-by-4 matrices, and coordinates can be represented by four-element row vectors, [*x, y, z,* 1].

If v_{child} is a coordinate in the child frame, then v_{parent}, the equivalent coordinate in the parent frame, is defined as:

$$v_{parent} = v_{child} T_{child}$$

T_{child} is the child frame's transformation matrix.

The transformations of all the parent frames above a child frame up to the root frame are concatenated with the transformation of that child to produce a world transformation. This world transformation is then applied to the visuals on the child frame before rendering. Coordinates relative to the child frame are sometimes called *model coordinates*. After the world transformation is applied, coordinates are called *world coordinates*.

The transformation of a frame can be modified directly by using the IDirect3DRMFrame::AddTransform, IDirect3DRMFrame::AddScale, IDirect3DRMFrame::AddRotation, and IDirect3DRMFrame::AddTranslation methods. Each of these methods specifies a member of the D3DRMCOMBINETYPE enumerated type, which specifies how the matrix supplied by the application should be combined with the current frame's matrix.

The IDirect3DRMFrame::GetRotation and IDirect3DRMFrame::GetTransform methods allow you to retrieve a frame's rotation axis and transformation matrix. To change the rotation of a frame, use the

IDirect3DRMFrame::SetRotation method.

Use the IDirect3DRMFrame::Transform and IDirect3DRMFrame::InverseTransform methods to change between world coordinates and frame coordinates.

Motion

Every frame has an intrinsic rotation and velocity. Frames that are neither rotating nor translating simply have zero values for these attributes. These attributes are used before each scene is rendered to move objects in the scene, and they can also be used to create simple animations.

Callback Functions

Frames support a callback function that you can use to support more complex animations. The application registers a function that the frame calls before the motion attributes are applied. Where there are multiple frames in a hierarchy, each with associated callback functions, the parent frames are called before the child frames. For a given hierarchy, rendering does not take place until all of the required callback functions have been invoked.

To add this callback function, use the IDirect3DRMFrame::AddMoveCallback method; to remove it, use the IDirect3DRMFrame::DeleteMoveCallback method.

You can use these callback functions to provide new positions and orientations from a preprogrammed animation sequence or to implement dynamic motion in which the activities of visuals depend upon the positions of other objects in the scene.

Direct3DRMLight and Direct3DRMLightArray

Lighting effects are employed to increase the visual fidelity of a scene. The system colors each object based on the object's orientation to the light sources in the scene. The contribution of each light source is combined to determine the color of the object during rendering. All lights have color and intensity that can be varied independently.

An application can attach lights to a frame to represent a light source in a scene. When a light is attached to a frame, it illuminates visual objects in the scene. The frame provides both position and orientation for the light. In other words, the light originates from the origin of the frame it is attached to. An application can move and redirect a light source simply by moving and reorienting the frame the light source is attached to.

Each viewport owns one or more lights. No light can be owned by more than one viewport. For more information about the interdependencies of Direct3D components, see Object Connectivity.

Retained Mode currently provides five types of light sources: ambient, directional, parallel point, point, and spotlight.

Ambient

An *ambient* light source illuminates everything in the scene, regardless of the orientation, position, and surface characteristics of the objects in the scene. Because ambient light illuminates a scene with equal strength everywhere, the position and orientation of the frame it is attached to are inconsequential. Multiple ambient light sources are combined within a scene.

Directional

A *directional* light source has orientation but no position. The light is attached to a frame but appears to illuminate all objects with equal intensity, as if it were at an infinite distance from the objects. The directional source is commonly used to simulate distant light sources, such as the sun. It is the best choice of light to use for maximum rendering speed.

Parallel Point

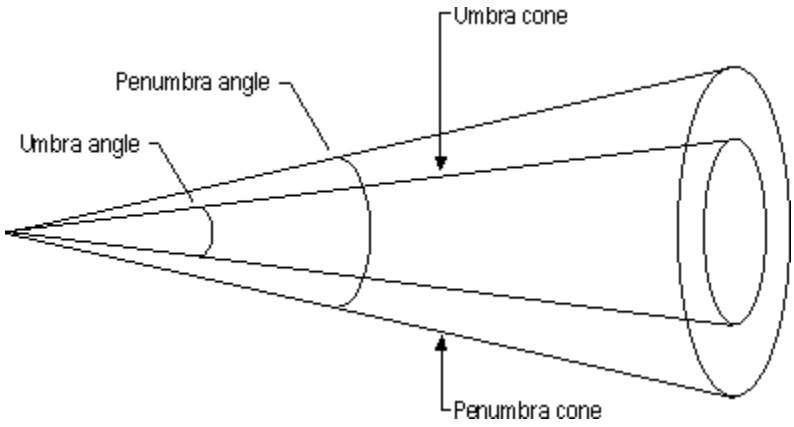
A *parallel point* light source illuminates objects with parallel light, but the orientation of the light is taken from the position of the parallel point light source. That is, like a directional light source, a parallel point light source has orientation, but it also has position. For example, two meshes on either side of a parallel point light source are lit on the side that faces the position of the source. The parallel point light source offers similar rendering-speed performance to the directional light source.

Point

A *point* light source radiates light equally in all directions from its origin. It requires the calculation of a new lighting vector for every facet or normal it illuminates, and for this reason it is computationally more expensive than a parallel point light source. It does, however, produce a more faithful lighting effect and should be chosen where visual fidelity is the deciding concern.

Spotlight

A *spotlight* emits a cone of light. Only objects within the cone are illuminated. The cone produces light of two degrees of intensity, with a central brightly lit section (the *umbra*) that acts as a point source, and a surrounding dimly lit section (the *penumbra*) that merges with the surrounding deep shadow. The angles of these two sections can be individually specified by using the IDirect3DRMLight::GetPenumbra, IDirect3DRMLight::GetUmbra, IDirect3DRMLight::SetPenumbra, and IDirect3DRMLight::SetUmbra methods.


Umbra cone
Penumbra angle
Umbra angle
Penumbra cone

Direct3DRMMaterial

A material defines how a surface reflects light. A material has two components: an emissive property (whether it emits light) and a specular property, whose brightness is determined by a power setting. The value of the power determines the sharpness of the reflected highlights, with a value of 5 giving a metallic appearance and higher values giving a more plastic appearance.

An application can control the emission of a material by using the IDirect3DRMMaterial::GetEmissive and IDirect3DRMMaterial::SetEmissive methods, the specular component by using the IDirect3DRMMaterial::GetSpecular and IDirect3DRMMaterial::SetSpecular methods, and the power by using the IDirect3DRMMaterial::GetPower and IDirect3DRMMaterial::SetPower methods.

Direct3DRMMesh and Direct3DRMMeshBuilder

A mesh is a visual object that is made up of a set of polygonal faces. A mesh defines a set of vertices and a set of faces (the faces are defined in terms of the vertices and normals of the mesh). Changing a vertex or normal that is used by several faces changes the appearance of all faces sharing it.

The vertices of a mesh define the positions of faces in the mesh, and they can also be used to define 2D coordinates within a texture map.

You can manipulate meshes in Retained Mode by using two COM interfaces: IDirect3DRMMesh and IDirect3DRMMeshBuilder. **IDirect3DRMMesh** is very fast, and you should use it when a mesh is subject to frequent changes, such as when morphing. **IDirect3DRMMeshBuilder** is built on top of the **IDirect3DRMMesh** interface. Although the **IDirect3DRMMeshBuilder** interface is a convenient way to perform operations on individual faces and vertices, the system must convert a Direct3DRMMeshBuilder object into a Direct3DRMMesh object before rendering it. For meshes that do not change or that change infrequently, this conversion has a negligible impact on performance.

If an application needs to assign the same characteristics (such as material or texture) to several vertices or faces, it can use the **IDirect3DRMMesh** interface to combine them in a group. If the application needs to share vertices between two different groups (for example, if neighboring faces in a mesh are different colors), the vertices must be duplicated in both groups. The IDirect3DRMMesh::AddGroup method assigns a group identifier to a collection of faces. This identifier is used to refer to the group in subsequent calls.

The **IDirect3DRMMeshBuilder** and **IDirect3DRMMesh** interfaces allow an application to create faces with more than three sides. They also automatically split a mesh into multiple buffers if, for example, the hardware the application is rendering to has a limit of 64K and a mesh is larger than that size. These features set the Direct3DRMMesh and Direct3DRMMeshBuilder API apart from the Direct3D API.

You can add vertices and faces individually to a mesh by using the IDirect3DRMMeshBuilder::AddVertex, IDirect3DRMMeshBuilder::AddFace, and IDirect3DRMMeshBuilder::AddFaces methods.

You can define individual color, texture, and material properties for each face in the mesh, or for all faces in the mesh at once, by using the IDirect3DRMMesh::SetGroupColor, IDirect3DRMMesh::SetGroupColorRGB, IDirect3DRMMesh::SetGroupTexture, and IDirect3DRMMesh::SetGroupMaterial methods.

For a mesh to be rendered, you must first add it to a frame by using the IDirect3DRMFrame::AddVisual method. You can add a single mesh to multiple frames to create multiple instances of that mesh.

Your application can use flat, Gouraud, and Phong shade modes, as specified by a call to the IDirect3DRMMesh::SetGroupQuality method. (Phong shading is not available for DirectX 2, however.) This method uses values from the D3DRMRENDERQUALITY enumerated type. For more information about shade modes, see Polygons.

You can set normals (which should be unit vectors), or normals can be calculated by averaging the face normals of the surrounding faces by using the IDirect3DRMMeshBuilder::GenerateNormals method.

Direct3DRMObject

Direct3DRMObject is the common superclass of all objects in the system. A Direct3DRMObject object has characteristics common to all objects.

Each Direct3DRMObject object is instantiated as a COM object. In addition to the methods of the IUnknown interface, each object has a standard set of methods that are generic to all.

To create an object, the application must first have instantiated a Direct3D Retained-Mode object by calling the Direct3DRMCreate function. The application then calls the method of the object's interface that creates an object, and it specifies parameters specific to the object. For example, to create a Direct3DRMAnimation object, the application would call the IDirect3DRM::CreateAnimation method. The creation method then creates a new object, initializes some of the object's attributes from data passed in the parameters (leaving all others with their default values), and returns the object. Applications can then specify the interface for this object to modify and use the object.

Any object can store 32 bits of application-specific data. This data is not interpreted or altered by Retained Mode. The application can read this data by using the IDirect3DRMObject::GetAppData method, and it can write to it by using the IDirect3DRMObject::SetAppData method. Finding this data is simpler if the application keeps a structure for each Direct3DRMFrame object. For example, if calling the IDirect3DRMFrame::GetParent method retrieves a Direct3DRMFrame object, the application can easily retrieve the data by using a pointer to its private structure, possibly avoiding a time-consuming search.

You might also want to assign a name to an object to help you organize an application or as part of your application's user interface. You can use the IDirect3DRMObject::SetName and IDirect3DRMObject::GetName methods to set and retrieve object names.

Another example of possible uses for application-specific data is when an application needs to group the faces within a mesh into subsets (for example, for front and back faces). You could use the application data in the face to note in which of these groups a face should be included.

An application can specify a function to call when an object is destroyed, such as when the application needs to deallocate memory associated with the object. To do this, use the IDirect3DRMObject::AddDestroyCallback method. To remove a function previously registered with this method, use the IDirect3DRMObject::DeleteDestroyCallback method.

The callback function is called only when the object is destroyed—that is, when the object's reference count has reached 0 and the system is about to deallocate the memory for the object. If an application kept additional data about an object (so that its dynamics could be implemented, for example), the application could use this callback function as a way to notify itself that it can dispose of the data.

For related information, see IDirect3DRMObject.

Direct3DRMPickedArray

Picking is the process of searching for visuals in a scene, given a 2D coordinate in a viewport. You can use the IDirect3DRMViewport::Pick method to retrieve an IDirect3DRMPickedArray interface, and then call the IDirect3DRMPickedArray::GetPick method to retrieve an IDirect3DRMFrameArray interface and a visual object. The array of frames is the path through the hierarchy leading to the visual object; that is, a hierarchical list of the visual object's parent frames, with the top-most parent in the hierarchy first in the array.

Direct3DRMShadow

Applications can produce an initialized and usable shadow simply by calling the IDirect3DRM::CreateShadow method. The IDirect3DRMShadow interface exists so that applications that create a shadow by using the IDirect3DRM::CreateObject method can initialize the shadow by calling the IDirect3DRMShadow::Init method.

Direct3DRMTexture

A texture is a rectangular array of colored pixels. (The rectangle does not necessarily have to be square, although the system deals most efficiently with square textures.) You can use textures for texture-mapping faces, in which case their dimensions must be powers of two.

An IDirect3DRMTexture interface is actually an interface to a DirectDrawSurface object, not to a distinct Direct3D texture object. For more information about the relationship between textures in Direct3D and surfaces in DirectDraw, see Direct3D Texture Interface.

Your application can use the IDirect3DRM::CreateTexture method to create a texture from a D3DRMIMAGE structure, or the IDirect3DRM::CreateTextureFromSurface method to create a texture from a DirectDraw surface. The IDirect3DRM::LoadTexture method allows your application to load a texture from a file; the texture should be in Windows bitmap (.bmp) or Portable Pixmap (.ppm) format.

The texture coordinates of each face define the region in the texture that is mapped onto that particular face. Your application can use a wrap to calculate texture coordinates. For more information, see Direct3DRMWrap.

Decals

Textures can also be rendered directly, as visuals. Textures used this way are sometimes known as *decals*, a term adopted by Retained Mode. A decal is rendered into a viewport-aligned rectangle. The rectangle can optionally be scaled by the depth component of the decal's position. The size of the decal is taken from a rectangle defined relative to the containing frame by using the IDirect3DRMTexture::SetDecalSize method. (An application can retrieve the size of the decal by using the IDirect3DRMTexture::GetDecalSize method.) The decal is then transformed and perspective projection is applied.

Decals have origins that your application can set and retrieve by using the IDirect3DRMTexture::SetDecalOrigin and IDirect3DRMTexture::GetDecalOrigin methods. The origin is an offset from the top-left corner of the decal. The default origin is [0, 0]. The decal's origin is aligned with its frame's position when rendering.

Texture Colors

You can set and retrieve the number of colors that are used to render a texture by using the IDirect3DRMTexture::SetColors and IDirect3DRMTexture::GetColors methods.

If your application uses the RGB color model, you can use 8-bit, 24-bit, and 32-bit textures. If you use the monochromatic (or ramp) color model, however, you can use only 8-bit textures.

Several shades of each color are used when lighting a scene. An application can set and retrieve the number of shades used for each color by calling the IDirect3DRMTexture::SetShades and IDirect3DRMTexture::GetShades methods.

A Direct3DRMTexture object uses a D3DRMIMAGE structure to define the bitmap that the texture will be rendered from. If the application provides the **D3DRMIMAGE** structure, the texture can easily be animated or altered during rendering.

Mipmaps

A mipmap is a sequence of textures, each of which is a progressively lower resolution, prefiltered representation of the same image. Mipmapping is a computationally low-cost way of improving the quality of rendered textures. Each prefiltered image, or level, in the mipmap is a power of two smaller than the previous level. You can specify mipmaps when filtering textures by calling the IDirect3DRMDevice::SetTextureQuality method.

For more information about how to use DirectDraw to create mipmaps, see Mipmaps.

Texture Filtering

After a texture has been mapped to a surface, the texture elements (texels) of the texture rarely correspond to individual pixels in the final image. A pixel in the final image can correspond to a large collection of texels or to a small piece of a single texel. You can use texture filtering to specify how to interpolate texel values to pixels.

You can use the IDirect3DRMDevice::SetTextureQuality method and the D3DRMTEXTUREQUALITY

enumerated type to specify the texture filtering mode for your application.

Texture Transparency

You can use the IDirect3DRMTexture::SetDecalTransparency method to produce transparent textures. Another method for achieving transparency is to use DirectDraw's support for *color keys*. Color keys are colors or ranges of colors that can be part of either the source or destination of a blit or overlay operation. You can specify that these colors should always be overwritten or never be overwritten.

For more information about DirectDraw's support for color keys, see Color Keying.

For related information, see IDirect3DRMTexture.

Direct3DRMUserVisual

User-visual objects are application-defined data that an application can add to a scene and then render, typically by using a customized rendering module. For example, an application could add sound as a user-visual object in a scene, and then render the sound during playback.

You can use the IDirect3DRM::CreateUserVisual method to create a user-visual object.

Direct3DRMViewport and Direct3DRMViewportArray

The viewport defines how the 3D scene is rendered into a 2D window. The viewport defines a rectangular area on a device that objects will be rendered into.

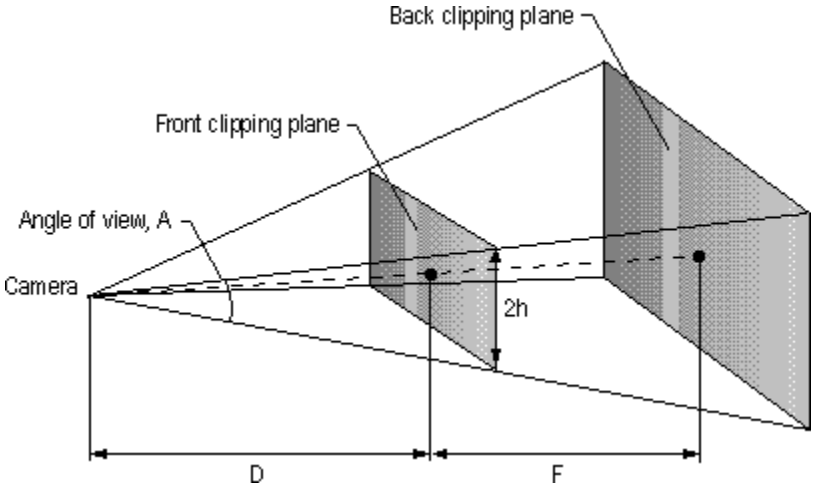
Camera

The viewport uses a Direct3DRMFrame object as a *camera*. The camera frame defines which scene is rendered and the viewing position and direction. The viewport renders only what is visible along the positive z-axis of the camera frame, with the up direction being in the direction of the positive y-axis.

An application can call the IDirect3DRMViewport::SetCamera method to set a camera for a given viewport. This method sets a viewport's position, direction, and orientation to that of the given camera frame. To retrieve the current camera settings, call the IDirect3DRMViewport::GetCamera method.

Viewing Frustum

The viewing frustum is a 3D volume in a scene positioned relative to the viewport's camera. Objects within the viewing frustum are visible. For perspective viewing, the viewing frustum is the volume of an imaginary pyramid that is between the front clipping plane and the back clipping plane.



The camera is at the tip of the pyramid, and its z-axis runs from the tip of the pyramid to the center of the pyramid's base. The front clipping plane is a distance *D* from the camera, and the back clipping plane is a distance *F* from the front clipping plane. An application can set and retrieve these values by using the IDirect3DRMViewport::SetFront, IDirect3DRMViewport::SetBack, IDirect3DRMViewport::GetFront, and IDirect3DRMViewport::GetBack methods. The height of the front clipping plane is 2*h* and defines the field of view. An application can set and retrieve the value of *h* by using the IDirect3DRMViewport::SetField and IDirect3DRMViewport::GetField methods.

The angle of view, *A*, is defined by the following equation, which can be used to calculate a value for *h* when a particular camera angle is desired:

$$A = 2 \tan^{-1} \frac{h}{D}$$

The viewing frustum is a pyramid only for perspective viewing. For orthographic viewing, the viewing frustum is cuboid. These viewing types (or projection types) are defined by the D3DRMPROJECTIONTYPE enumerated type and used by the IDirect3DRMViewport::GetProjection and IDirect3DRMViewport::SetProjection methods.

Transformations

To render objects with 3D coordinates in a 2D window, the object must be transformed into the camera's frame. A projection matrix is then used to give a four-element homogeneous coordinate [*x y z w*], which is used to derive a three-element coordinate [*x/w y/w z/w*], where [*x/w y/w*] is the coordinate to be used in the window and *z/w* is the depth, ranging from 0 at the front clipping plane to 1 at the back clipping plane. The projection matrix is a combination of a perspective transformation followed by a scaling and translation to scale the objects into the window.

The following matrix is the projection matrix. In these formulas, *h* is the half-height of the viewing

frustum, *F* is the position in z-coordinates of the back clipping plane, and *D* is the position in z-coordinates of the front clipping plane:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{hF}{D(F-D)} & \frac{h}{D} \\ 0 & 0 & \frac{-hF}{F-D} & 0 \end{bmatrix}$$

The following matrix is the window-scaling matrix. (The scales are dependent on the size and position of the window.) In these formulas, *s* is the window-scaling factor and *o* is the window origin:

$$W = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & -sy & 0 & 0 \\ 0 & 0 & 1 & 0 \\ ox & oy & 0 & 1 \end{bmatrix}$$

The following matrix is the viewing matrix. This is a combination of the projection matrix and window matrix, or the dot product of P and W:

$$V = PW = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & -sy & 0 & 0 \\ \frac{hox}{D} & \frac{hoy}{D} & \frac{hF}{D(F-D)} & \frac{h}{D} \\ 0 & 0 & \frac{-hF}{F-D} & 0 \end{bmatrix}$$

The scaling factors and origin s_x, s_y, o_x, and o_y are chosen so that the region [*-h -h D*] to [*h h D*] fits exactly into either the window's height or the window's width, whichever is larger.
The application can use the IDirect3DRMViewport::Transform and IDirect3DRMViewport::InverseTransform methods to transform vectors to screen coordinates from world coordinates and vice versa. An application can use these methods to support dragging, as shown in the following example:

```
/*
 * Drag a frame by [delta_x delta_y] pixels in the view.
 */
void DragFrame(LPDIRECT3DRMVIEWPORT view,
    LPDIRECT3DRMFRAME frame,
    LPDIRECT3DRMFRAME scene,
    int delta_x, int delta_y)
{
    D3DVECTOR p1;
    D3DRMVECTOR4D p2;

    frame->GetPosition(scene, &p1);
    view->Transform(&p2, &p1);
    p2.x += delta_x * p2.w;
    p2.y += delta_y * p2.w;
    view->InverseTransform(&p1, &p2);
    frame->SetPosition(scene, p1.x, p1.y, p1.z);
}
```

An application uses the viewport transformation to ensure that the distance by which the object is moved in world coordinates is scaled by the object's distance from the camera to account for perspective. Note that the result from IDirect3DRMViewport::Transform is a four-element homogeneous vector. This avoids the problems associated with coordinates being scaled by an infinite amount near the camera's position.

The viewport projection matrix produces a well-defined 3D coordinate only for points inside the viewing

frustum. For a homogeneous point [*x y z w*] after projection, this is true if the following equations hold:

$$wx_{min} \le x < wx_{max}$$
$$wy_{min} \le y < wy_{max}$$
$$0 \le z < w$$

where

$$x_{min} = viewport_x - viewport_{width} / 2$$
$$x_{max} = viewport_x + viewport_{width} / 2$$
$$y_{min} = viewport_y - viewport_{height} / 2$$
$$y_{max} = viewport_y + viewport_{height} / 2$$

Picking

Picking is the process of searching for visuals in the scene given a 2D coordinate in the viewport's window. An application can use the IDirect3DRMViewport::Pick method to retrieve either the closest object in the scene or a depth-sorted list of objects.

Direct3DRMVisual and Direct3DRMVisualArray

Visuals are objects that can be rendered in a scene. Visuals are visible only when they are added to a frame in that scene. An application can add a visual to a frame by using the IDirect3DRMFrame::AddVisual method. The frame provides the visual with position and orientation for rendering.

You should use the IDirect3DRMVisualArray interface to work with groups of visual objects; although there is a **IDirect3DRMVisual** COM interface, it has no methods.

The most common visual types are Direct3DRMMeshBuilder and Direct3DRMTexture objects.

Direct3DRMWrap

You can use a wrap to calculate texture coordinates for a face or mesh. To create a wrap, the application must specify a type, a reference frame, an origin, a direction vector, and an up vector. The application must also specify a pair of scaling factors and an origin for the texture coordinates.

Your application calls the IDirect3DRM::CreateWrap function to create an IDirect3DRMWrap interface. This interface has two unique methods: IDirect3DRMWrap::Apply, which applies a wrap to the vertices of the object, and IDirect3DRMWrap::ApplyRelative, which transforms the vertices of a wrap as it is applied.

The D3DRMMAPPING type includes the D3DRMMAP_WRAPU and D3DRMMAP_WRAPV flags. These flags determine how the rasterizer interprets texture coordinates. The rasterizer always interpolates the shortest distance between texture coordinates—that is, a line. The path taken by this line, and the valid values for the u- and v-coordinates, varies with the use of the wrapping flags. If either or both flags is set, the line can wrap around the texture edge in the u or v direction, as if the texture had a cylindrical or toroidal topology.

- In flat wrapping mode, in which neither of the wrapping flags is set, the plane specified by the u- and v-coordinates is an infinite tiling of the texture. In this case, values greater than 1.0 are valid for u and v. The shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0.5, 0.5).
- If either D3DRMMAP_WRAPU or D3DRMMAP_WRAPV is set, the texture is a cylinder with an infinite length and a circumference of 1.0. Texture coordinates greater than 1.0 are valid only in the dimension that is not wrapped. The shortest distance between texture coordinates varies with the wrapping flag; if D3DRMMAP_WRAPU is set, the shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0, 0.5).
- If both D3DRMMAP_WRAPU and D3DRMMAP_WRAPV are set, the texture is a torus. Because the system is closed, texture coordinates greater that 1.0 are invalid. The shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0, 0).

Although texture coordinates that are outside the valid range may be truncated to valid values, this behavior is not defined.

Typically, applications set a wrap flag for cylindrical wraps when the intersection of the texture edges does not match the edges of the face; applications do not set a wrap flag when more than half of a texture is applied to a single face.

Wrapping Types

There are four wrapping types:

- Flat
- Cylindrical
- Spherical
- Chrome

This section describes these wrapping types. In the examples, the direction vector (the v vector) lies along the z-axis, and the up vector (the u vector) lies along the y-axis, with the origin at [0 0 0].

Flat

The flat wrap conforms to the faces of an object as if the texture were a piece of rubber that was stretched over the object.

The [*u v*] coordinates are derived from a vector [*x y z*] by using the following equations:

$u = s_u x - o_u$

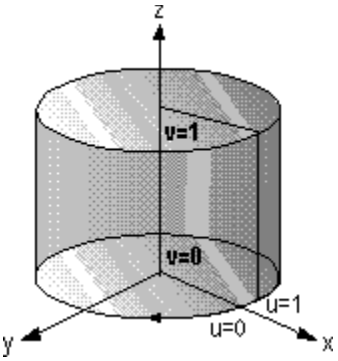
$v = s_v y - o_v$

In these formulas, *s* is the window-scaling factor and *o* is the window origin. The application should choose a pair of scaling factors and offsets that map the ranges of *x* and *y* to the range from 0 to 1 for *u* and *v*.

Cylindrical

The cylindrical wrap treats the texture as if it were a piece of paper that is wrapped around a cylinder so that the left edge is joined to the right edge. The object is then placed in the middle of the cylinder and the texture is deformed inward onto the surface of the object.

For a cylindrical texture map, the effects of the various vectors are shown in the following illustration.



The direction vector specifies the axis of the cylinder, and the up vector specifies the point on the outside of the cylinder where *u* equals 0. To calculate the texture [*u v*] coordinates for a vector [*x y z*], the system uses the following equations:

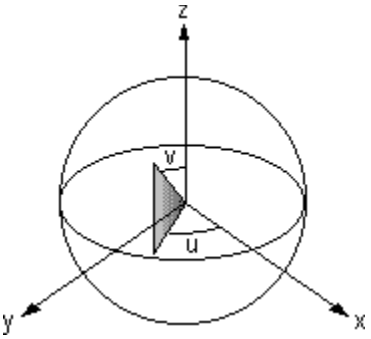
$$u = \frac{s_u}{2\pi} \tan^{-1}\frac{x}{y} - o_y$$

$$v = s_v z - o_v$$

Typically, *u* would be left unscaled and *v* would be scaled and translated so that the range of *z* maps to the range from 0 to 1 for *v*.

Spherical

For a spherical wrap, the u-coordinate is derived from the angle that the vector [*x y* 0] makes with the x-axis (as in the cylindrical map) and the v-coordinate from the angle that the vector [*x y z*] makes with the z-axis. Note that this mapping causes distortion of the texture at the z-axis.



This translates to the following equations:

$$u = \frac{S_u}{2\pi} \tan^{-1}\frac{x}{y} - o_u$$

$$v = \frac{S_v}{\pi} \tan^{-1}\frac{z}{\sqrt{x^2+y^2+z^2}} - o_v$$

The scaling factors and texture origin will often not be needed here as the unscaled range of *u* and *v* is already 0 through 1.

Chrome

A chrome wrap allocates texture coordinates so that the texture appears to be reflected onto the objects. The chrome wrap takes the reference frame position and uses the vertex normals in the mesh to calculate reflected vectors. The texture u- and v-coordinates are then calculated from the intersection of these reflected vectors with an imaginary sphere that surrounds the mesh. This gives the effect of the mesh reflecting whatever is wrapped on the sphere.

Direct3D Retained-Mode Tutorial

To create a Windows-based Direct3D Retained-Mode application, you set up the features of two different environments: the devices, viewports, and color capabilities of the Windows environment, and the models, textures, lights, and positions of the virtual environment. This section describes some simple Retained-Mode sample code that is part of this SDK.

Setting up the Windows Environment

The Rmmain.cpp file in the samples provded with this SDK is used as a basis for all of the Retained-Mode samples. This file contains the standard Windows framework of initialization, setting up a message loop, and creating a window procedure for message processing, but it also does some work that is specific to Direct3D Retained-Mode applications:

- It enumerates the current Direct3D device drivers. This is described in the Enumerating Direct3D Device Drivers section.
- It creates the Retained-Mode API, the scene and camera frames, and a DirectDrawClipper object. This is described in the Initialization section.
- It creates a Direct3DRM device and viewport. This is described in the Creating a Direct3DRM Device and Viewport section.
- It sets the rendering quality, dithering flag, texture quality, shades, default texture colors, and default texture shades. This is described in the Setting the Render State section.
- It calls a locally defined BuildScene function to set up the lights and visuals in a scene. This is described in the Setting Up the Virtual Environment section.
- It renders the scene. This is described in the Rendering into a Viewport section.
- It makes the mouse status available to sample applications. This is described in the OverrideDefaults and ReadMouse section.

Enumerating Direct3D Device Drivers

Rmmain.cpp calls the locally defined InitApp function as its first action inside the WinMain function, and it fails if InitApp is not successful. InitApp creates the window and initializes all objects that are required to begin rendering a 3D scene.

After performing some standard tasks in the initialization of a Windows application and initializing global variables to acceptable default settings, InitApp verifies that Direct3D device drivers are present by calling the locally defined EnumDrivers function.

EnumDrivers sets up a callback function that examines each of the drivers in the system and picks the best one for the application's needs. This is not the easiest or even the best way to find Direct3D device drivers, however. Instead of setting up an enumeration routine, most applications will allow the system to automatically enumerate the current drivers by calling the IDirect3DRM::CreateDeviceFromClipper method and specifying NULL for the *lpGUID* parameter. This is the recommended way to create a Retained-Mode device because it always works, even if the user installs new hardware. When you use this method and both a hardware and a software device meet the default requirements, the system always retrieves the hardware device. An application should enumerate devices instead of specifying NULL for *lpGUID* only if it has unusual requirements.

EnumDrivers first calls the DirectDrawCreate function to create a DirectDraw object and then calls IDirectDraw::QueryInterface to retrieve an interface to Direct3D. The first parameter of this method is the interface identifier (IID) that identifies the IDirect3D COM interface. These identifiers are defined in the D3d.h header file. Each identifier is in the form IID_*InterfaceName*. For example, the IID for the **IDirect3DRMTexture** interface is IID_IDirect3DRMTexture.

After creating the Direct3D interface, EnumDrivers calls the IDirect3D::EnumDevices method to enumerate the Direct3D drivers. The first parameter to this method is a D3DENUMDEVICESCALLBACK callback function, in this case named enumDeviceFunc. The second parameter is a pointer to the myglobs structure, which contains the global variables Rmmain.cpp uses to set up the 3D environment. The **CurrDriver** member of the myglobs structure is simply an integer indicating the number of the Direct3D driver currently being used. The application initializes this value to -1 before calling **IDirect3D::EnumDevices**; the callback function checks this value to find out whether the driver being enumerated is the first valid driver.

```
static BOOL EnumDrivers(HWND win)
{
LPDIRECTDRAW lpDD;
LPDIRECT3D lpD3D;
HRESULT rval;
HMENU hmenu;

rval = DirectDrawCreate(NULL, &lpDD, NULL);
rval = lpDD->QueryInterface(IID_IDirect3D, (void**) &lpD3D);

myglobs.CurrDriver = -1;
rval = lpD3D->EnumDevices(enumDeviceFunc, &myglobs.CurrDriver);

lpD3D->Release();
lpDD->Release();

// Code omitted here that adds the driver names to the File menu.

return TRUE;
}
```

The enumDeviceFunc (D3DENUMDEVICESCALLBACK) callback function records each usable Direct3D device driver's name and globally unique identifier (GUID) and chooses a driver. The following code sample shows the enumDeviceFunc callback function. First, this callback function creates a D3DDEVICEDESC structure and copies the input *lpContext* parameter to a parameter called *lpStartDriver*. (The *lpContext* parameter is application-defined data—it can be any data the application requires.) The fourth parameter passed to enumDeviceFunc is a pointer to a **D3DDEVICEDESC** structure describing the hardware; the function uses the **dcmColorModel** member of this structure to

determine whether it should examine the description of the hardware or the description of the hardware emulation layer (HEL). Any driver that cannot support the current bit-depth is skipped. The current driver's GUID and name are copied into the myglobs structure. Then the enumDeviceFunc callback function performs a few tests to compare the driver being enumerated against the driver stored as *lpStartDriver*. Whenever the driver being enumerated is implemented in hardware or uses the RGB instead of the monochromatic color model, that driver becomes the new *lpStartDriver*.

Finally, enumDeviceFunc increments the variable that counts the number of drivers. If this count exceeds the maximum, it returns D3DENUMRET_CANCEL to cancel the enumeration. Otherwise, it returns D3DENUMRET_OK to continue the enumeration.

```
static HRESULT WINAPI enumDeviceFunc(
    LPGUID lpGuid, LPSTR lpDeviceDescription, LPSTR lpDeviceName,
    LPD3DDEVICEDESC lpHWDesc, LPD3DDEVICEDESC lpHELDesc,
    LPVOID lpContext)
{
static BOOL hardware = FALSE; // Current start driver is hardware
static BOOL mono = FALSE;     // Current start driver is mono light
LPD3DDEVICEDESC lpDesc;
int *lpStartDriver = (int *)lpContext;

// Decide which device description we should consult.

lpDesc = lpHWDesc->dcmColorModel ? lpHWDesc : lpHELDesc;

// If this driver cannot render in the current display bit depth, skip
// it and continue with the enumeration.

if (!(lpDesc->dwDeviceRenderBitDepth & BPPToDDBD(myglobs.BPP)))
    return D3DENUMRET_OK;

// Record this driver's info.

memcpy(&myglobs.DriverGUID[myglobs.NumDrivers], lpGuid, sizeof(GUID));
strcpy(&myglobs.DriverName[myglobs.NumDrivers][0], lpDeviceName);

// Choose hardware over software, RGB lights over mono lights.

if (*lpStartDriver == -1) {

    // This is the first valid driver.

    *lpStartDriver = myglobs.NumDrivers;
    hardware = lpDesc == lpHWDesc ? TRUE : FALSE;
    mono = lpDesc->dcmColorModel & D3DCOLOR_MONO ? TRUE : FALSE;
} else if (lpDesc == lpHWDesc && !hardware) {

    // This driver is hardware and start driver is not.

    *lpStartDriver = myglobs.NumDrivers;
    hardware = lpDesc == lpHWDesc ? TRUE : FALSE;
    mono = lpDesc->dcmColorModel & D3DCOLOR_MONO ? TRUE : FALSE;
} else if ((lpDesc == lpHWDesc && hardware ) || (lpDesc == lpHELDesc
                        && !hardware)) {
    if (lpDesc->dcmColorModel == D3DCOLOR_MONO && !mono) {

    // This driver and start driver are the same type, and this
    // driver is mono while start driver is not.

    *lpStartDriver = myglobs.NumDrivers;
    hardware = lpDesc == lpHWDesc ? TRUE : FALSE;
```

```
    mono = lpDesc->dcmColorModel & D3DCOLOR_MONO ? TRUE : FALSE;
    }
}
myglobs.NumDrivers++;
if (myglobs.NumDrivers == MAX_DRIVERS)
    return (D3DENUMRET_CANCEL);
return (D3DENUMRET_OK);
}
```

When EnumDrivers function returns, it calls the IDirect3D::Release and IDirectDraw::Release methods to decrease the reference count of these COM objects. Every time an application calls the **Release** method on an object, the reference count for that object is reduced by 1. The system deallocates the object when its reference count reaches 0.

Whenever Rmmain.cpp calls a COM method, the return value is checked and the function fails if the method does not succeed. This error-checking has been omitted from the samples in this section for the sake of brevity. Sometimes the error-checking code calls the D3DRMErrorToString function (which is implemented in Rmerror.c in the same directory as Rmmain.cpp). This function simply maps an **HRESULT** error value to an explanatory string and returns the string.

If any COM interfaces have already been successfully created when a method fails, the error-checking code calls the **Release** method for that interface before returning FALSE.

Initialization

After enumerating the Direct3D device drivers, the InitApp function in Rmmain.cpp calls the Direct3DRMCreate function to create the Retained-Mode API:

```
LPDIRECT3DRM lpD3DRM;
HRESULT rval;

rval = Direct3DRMCreate(&lpD3DRM);
```

The sample then uses the Direct3DRM object to create the master reference frame (the "scene") and the camera frame by calling the IDirect3DRM::CreateFrame method. After creating the camera frame, this sample calls the IDirect3DRMFrame::SetPosition method to set its position.

```
rval = lpD3DRM->CreateFrame(NULL, &myglobs.scene);
rval = lpD3DRM->CreateFrame(myglobs.scene, &myglobs.camera);

// Set the position of the camera frame.

rval = myglobs.camera->SetPosition(myglobs.scene, D3DVAL(0.0),
    D3DVAL(0.0), D3DVAL(0.0));
```

The **scene** and **camera** members of the myglobs structure have the same type: **LPDIRECT3DRMFRAME**. The D3DVAL macro creates a value whose type is D3DVALUE from a supplied floating-point number.

Next, InitApp creates a DirectDraw clipper object by calling the DirectDrawCreateClipper function, and associates the window with the clipper object by calling the IDirectDrawClipper::SetHWnd method. This clipper object is subsequently used to create the Direct3D Windows device. The advantage of using a clipper object to create the device is that in this case DirectDraw does the clipping automatically.

```
LPDIRECTDRAWCLIPPER lpDDClipper;
HWND win;

rval = DirectDrawCreateClipper(0, &lpDDClipper, NULL);
rval = lpDDClipper->SetHWnd(0, win);
```

InitApp uses the DirectDraw clipper object and the Direct3D driver retrieved by the EnumDrivers function in a call to the locally defined CreateDevAndView function. CreateDevAndView is discussed in the following section, Creating a Direct3DRM Device and Viewport.

```
GetClientRect(win, &rc);
if (!CreateDevAndView(lpDDClipper, myglobs.CurrDriver,
        rc.right, rc.bottom)) {
    return FALSE;
}

// Create the scene to be rendered by calling this sample's BuildScene.

if (!BuildScene(myglobs.dev, myglobs.view, myglobs.scene,
    myglobs.camera))
    return FALSE;

// Ready to render.

myglobs.bInitialized = TRUE;

// Display the window.

ShowWindow(win, cmdshow);
UpdateWindow(win);

return TRUE;
```

Creating a Direct3DRM Device and Viewport

The InitApp function calls the locally defined CreateDevAndView function to create a Direct3DRM device and viewport. This function's parameters are a pointer to a DirectDrawClipper object, the number identifying the current driver, and the width and height of the current window's client rectangle.

After verifying that the window dimensions are valid, CreateDevAndView calls the IDirect3DRM::CreateDeviceFromClipper method to create a Direct3DRM device (stored in the **dev** member of the myglobs structure).

```
static BOOL CreateDevAndView(LPDIRECTDRAWCLIPPER lpDDClipper,
    int driver, int width, int height)
{
HRESULT rval;

if (!width || !height) {
    Msg("Cannot create a D3DRM device with invalid window dimensions.");
    return FALSE;
}

// Create the D3DRM device from this window and using the specified D3D
// driver.

rval = lpD3DRM->CreateDeviceFromClipper(lpDDClipper,
    &myglobs.DriverGUID[driver], width, height, &myglobs.dev);
```

CreateDevAndView uses the device it just created and the camera frame in a call to the IDirect3DRM::CreateViewport method that creates a Direct3DRM viewport. The width and height of the viewport are retrieved by calling the IDirect3DRMDevice::GetWidth and IDirect3DRMDevice::GetHeight methods. The viewport is stored in the **view** member of the myglobs structure. After creating the viewport, the code sets its background depth to a large number by calling the IDirect3DRMViewport::SetBack method.

```
width = myglobs.dev->GetWidth();
height = myglobs.dev->GetHeight();
rval = lpD3DRM->CreateViewport(myglobs.dev, myglobs.camera, 0, 0, width,
    height, &myglobs.view);

rval = myglobs.view->SetBack(D3DVAL(5000.0));
```

Finally, CreateDevAndView calls the locally defined SetRenderState function to set the rendering quality, fill mode, lighting state, and color shade. This function is described in Setting the Render State.

```
// Set the rendering quality, fill mode, lighting state and
// color shade information.

if (!SetRenderState())
    return FALSE;
return TRUE;
}
```

Setting the Render State

The InitApp function calls CreateDevAndView, which in turn calls the SetRenderState function to set the rendering quality, fill mode, lighting state, and color shade. Rmmain.cpp calls the SetRenderState function whenever the render state may have changed; for example, whenever the user changes the fill or lighting modes.

SetRenderState calls the IDirect3DRMDevice::SetQuality method to set the rendering quality, IDirect3DRMDevice::SetDither to set the dithering flag, and IDirect3DRMDevice::SetTextureQuality to set the texture quality. Each of these methods has a **Get** counterpart, which SetRenderState calls before setting any values. If the current value in the myglobs structure is the same as the value retrieved by the **Get** method, the function takes no further action.

```
BOOL SetRenderState(void)
{
HRESULT rval;

if (myglobs.dev->GetQuality() != myglobs.RenderQuality) {
    rval = myglobs.dev->SetQuality(myglobs.RenderQuality);
}

if (myglobs.dev->GetDither() != myglobs.bDithering) {
    rval = myglobs.dev->SetDither(myglobs.bDithering);
}

if (myglobs.dev->GetTextureQuality() != myglobs.TextureQuality) {
    rval = myglobs.dev->SetTextureQuality(myglobs.TextureQuality);
}
```

Now SetRenderState sets the shades, default texture colors, and default texture shades based on the current number of bits per pixel.

- If the device supports only one bit per pixel, SetRenderState uses IDirect3DRMDevice::SetShades to set the number of shades to four; this is the number of shades in a ramp of colors used for shading. SetRenderState also calls IDirect3DRM::SetDefaultTextureShades to set the default number of shades for textures.
- If the device supports 16 bits per pixel, SetRenderState calls an additional method: IDirect3DRM::SetDefaultTextureColors. This method sets the default number of colors used in textures.
- If the device supports 24 or 32 bits per pixel, SetRenderState calls the same methods as for 16 bits per pixel. The only difference is that in this case it sets the shades and default texture shades to 256 instead of 32.

```
switch (myglobs.BPP) {
    case 1:
    if (FAILED(myglobs.dev->SetShades(4)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureShades(4)))
        goto shades_error;
    break;
    case 16:
    if (FAILED(myglobs.dev->SetShades(32)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureColors(64)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureShades(32)))
        goto shades_error;
    break;
    case 24:
    case 32:
    if (FAILED(myglobs.dev->SetShades(256)))
```

```
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureColors(64)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureShades(256)))
        goto shades_error;
    break;
}
return TRUE;

shades_error:
Msg("A failure occurred while setting color shade information.\n");
return FALSE;
}
```

The FAILED macro is defined as follows in the Winerror.h header file:

```
#define FAILED(Status) ((HRESULT)(Status)<0)
```

Setting Up the Virtual Environment

After the InitApp function has created the window, enumerated the Direct3D drivers, created the Retained-Mode API, created the scene and camera frames, created a DirectDrawClipper object, and created the Direct3D device and viewport, it calls a locally defined BuildScene function to build the scene. Each of the Direct3D samples in this SDK includes its own BuildScene function. These implementations of BuildScene vary depending on the capabilities being demonstrated by the sample.

This section discusses the simplest implementation of BuildScene in this SDK: the version in Egg.c. In this module, BuildScene performs the following tasks:

- Sets up the lights. This is described in the Setting Up the Lights section.
- Adds a mesh to the scene and sets up the position, rotation, and orientation of its frame. This is described in the Loading and Adding a Mesh section.
- Releases the objects that it has created. This is described in the Releasing the Direct3DRM Objects section.
- Overrides several of the default settings from Rmmain.cpp. This is described in the OverrideDefaults and ReadMouse section.

Setting Up the Lights

The BuildScene function implements two light sources: an ambient light and a directional light. An ambient light casts a uniform light over the entire scene. The position and orientation of its containing frame are ignored. A directional light, on the other hand, has direction but no position. It is attached to a frame but appears to illuminate all objects with equal intensity, as if it were at an infinite distance from each object.

First the BuildScene function creates a frame for the directional light and calls the IDirect3DRMFrame::SetPosition method to set its position in the scene. Then it calls the IDirect3DRM::CreateLightRGB method to create a directional light source. The IDirect3DRMFrame::AddLight method adds the directional light to the lighting frame. Then the IDirect3DRMFrame::SetRotation method causes the lighting frame to rotate.

After setting up the directional light, BuildScene creates a much dimmer ambient light source, again by calling **IDirect3DRM::CreateLightRGB**. Because ambient lights are not affected by the position and orientation of their containing frame, the BuildScene function specifies the frame for the entire scene in the call to **IDirect3DRMFrame::AddLight** that adds the ambient light.

```
LPDIRECT3DRMFRAME lights = NULL;
LPDIRECT3DRMLIGHT light1 = NULL;
LPDIRECT3DRMLIGHT light2 = NULL;
    .
    .
    .
if (FAILED(lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, scene, &lights)))
    goto generic_error;
if (FAILED(lights->lpVtbl->SetPosition(lights, scene, D3DVAL(5),
        D3DVAL(5), -D3DVAL(1))))
    goto generic_error;
if (FAILED(lpD3DRM->lpVtbl->CreateLightRGB(lpD3DRM,
        D3DRMLIGHT_DIRECTIONAL, D3DVAL(0.9),
        D3DVAL(0.8), D3DVAL(0.7), &light1)))
    goto generic_error;
if (FAILED(lights->lpVtbl->AddLight(lights, light1)))
    goto generic_error;
if (FAILED(lights->lpVtbl->SetRotation(lights, scene, D3DVAL(0),
        D3DVAL(1), D3DVAL(1), D3DVAL(-0.02))))
    goto generic_error;
if (FAILED(lpD3DRM->lpVtbl->CreateLightRGB(lpD3DRM, D3DRMLIGHT_AMBIENT,
        D3DVAL(0.1), D3DVAL(0.1), D3DVAL(0.1), &light2)))
    goto generic_error;
if (FAILED(scene->lpVtbl->AddLight(scene, light2)))
    goto generic_error;
```

The generic_error label identifies a place in the code where Egg.c displays a message box with an error value, releases any created COM objects, and returns FALSE from the BuildScene function.

Loading and Adding a Mesh

The BuildScene function in Egg.c loads a simple egg-shaped mesh into the scene and rotates the frame containing the mesh.

A call to the IDirect3DRM::CreateMeshBuilder method retrieves the IDirect3DRMMeshBuilder COM interface. BuildScene uses this interface for only one call before releasing it; it calls the IDirect3DRMMeshBuilder::Load method to load a mesh file named Egg.x.

Next, BuildScene calls IDirect3DRM::CreateFrame to create a frame for the mesh, and then it calls IDirect3DRMFrame::AddVisual to add the mesh to that frame.

Then, BuildScene sets the position and orientation of the camera frame in relation to the scene frame by calling the IDirect3DRMFrame::SetPosition and IDirect3DRMFrame::SetOrientation methods. Finally, it rotates the mesh frame in relation to the scene frame by calling the IDirect3DRMFrame::SetRotation method.

```
LPDIRECT3DRMMESHBUILDER egg_builder = NULL;
LPDIRECT3DRMFRAME egg = NULL;
    .
    .
    .
if (FAILED(lpD3DRM->lpVtbl->CreateMeshBuilder(lpD3DRM, &egg_builder)))
    goto generic_error;
rval = egg_builder->lpVtbl->Load(egg_builder, "egg.x", NULL,
    D3DRMLOAD_FROMFILE, NULL, NULL);

if (FAILED(lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, scene, &egg)))
    goto generic_error;

if (FAILED(egg->lpVtbl->AddVisual(
        egg, (LPDIRECT3DRMVISUAL)egg_builder)))
    goto generic_error;

if (FAILED(camera->lpVtbl->SetPosition(
        camera, scene, D3DVAL(0), D3DVAL(0), -D3DVAL(10))))
    goto generic_error;
if (FAILED(camera->lpVtbl->SetOrientation(
        camera, scene, D3DVAL(0), D3DVAL(0), D3DVAL(1), D3DVAL(0),
        D3DVAL(1), D3DVAL(0))))
    goto generic_error;
if (FAILED(egg->lpVtbl->SetRotation(
        egg, scene, D3DVAL(0), D3DVAL(1), D3DVAL(1), D3DVAL(0.02))))
    goto generic_error;
```

Releasing the Direct3DRM Objects

The last task performed by the BuildScene function in Egg.c is to release the Direct3DRM objects it creates—the frames for the mesh and lights, the mesh itself, and the two lights:

```
RELEASE(egg);
RELEASE(lights);
RELEASE(egg_builder);
RELEASE(light1);
RELEASE(light2);
```

The RELEASE macro calls the **Release** method for the appropriate interface. It is defined in C++ and C versions in the Rmdemo.h header file. The syntax of the C version looks like this:

```
#define RELEASE(x) if (x != NULL) {x->lpVtbl->Release(x); x = NULL;}
```

Rmmain.cpp also releases any objects it creates, in its CleanUpAndPostQuit function. The code calls CleanUpAndPostQuit whenever initialization has been completed and the application must exit: when a WM_QUIT message is received, when the user chooses the Exit command, on fatal errors, and so on.

```
void CleanUpAndPostQuit(void)
{
myglobs.bInitialized = FALSE;
RELEASE(myglobs.scene);
RELEASE(myglobs.camera);
RELEASE(myglobs.view);
RELEASE(myglobs.dev);
RELEASE(lpD3DRM);
RELEASE(lpDDClipper);
myglobs.bQuit = TRUE;
}
```

OverrideDefaults and ReadMouse

Samples can use the OverrideDefaults function with Rmmain.cpp, in addition to the standard BuildScene function. Egg.c includes an implementation of OverrideDefaults that simply sets the **bNoTextures** member to TRUE in the Defaults structure (defined in Rmdemo.h) and changes the name of the application:

```
void OverrideDefaults(Defaults *defaults)
{
    defaults->bNoTextures = TRUE;
    strcpy(defaults->Name, "Egg Direct3DRM Example");
}
```

Rmmain.cpp includes another function, ReadMouse, that samples can use to retrieve the mouse status. For an example of how this function is used, see the Quat.c sample in this SDK.

```
void ReadMouse(int* b, int* x, int* y)
{
    *b = myglobs.mouse_buttons;
    *x = myglobs.mouse_x;
    *y = myglobs.mouse_y;
}
```

Rendering into a Viewport

The Rmmain.cpp file calls the RenderLoop function as part of the message loop in the WinMain function. The RenderLoop function performs a few simple tasks:

- Calls IDirect3DRMFrame::Move to apply the rotations and velocities for all frames in the hierarchy.
- Calls IDirect3DRMViewport::Clear to clear the current viewport, setting it to the current background color.
- Calls IDirect3DRMViewport::Render to render the current scene into the current viewport.
- Calls IDirect3DRMDevice::Update to copy the rendered image to the display.

```
static BOOL RenderLoop()
{
HRESULT rval;
rval = myglobs.scene->Move(D3DVAL(1.0));
rval = myglobs.view->Clear();
rval = myglobs.view->Render(myglobs.scene);
rval = myglobs.dev->Update();
return TRUE;
}
```

About Immediate Mode

This section describes Direct3D's Immediate Mode, Microsoft's low-level 3D API. Direct3D's Immediate Mode is ideal for developers who need to port games and other high-performance multimedia applications to the Microsoft Windows operating system. Immediate Mode is a device-independent way for applications to communicate with accelerator hardware at a low level. Direct3D's Retained Mode is built on top of Immediate Mode.

Developers who use Immediate Mode instead of Retained Mode are typically experienced in high-performance programming issues, and may also be experienced in 3D graphics. Even if this is the case for you, you should read A Technical Foundation for 3D Programming. This section discusses implementation details of Direct3D that you need to know to work effectively with the system. The overall Direct3D architecture is described in Direct3D Architecture; this is essential reading for Immediate-Mode developers. If you want an overview of Immediate Mode, you should read Introduction to Direct3D Immediate-Mode Objects. Your best source of information about Immediate Mode, however, is probably the sample code included with this SDK; it illustrates how to put Direct3D's Immediate Mode to work in real-world applications.

This section is not an introduction to programming with Direct3D's Immediate Mode; for this information, see Direct3D Immediate-Mode Tutorial.

Introduction to Direct3D Immediate-Mode Objects

Direct3D's Immediate Mode consists of API elements that create objects, fill them with data, and link them together. Direct3D's Retained Mode is built on top of Immediate Mode. For a description of the overall organization of the system and the organization of Immediate Mode in particular, see Direct3D Architecture.

The following table shows the eight object types in Immediate Mode, their Component Object Model (COM) interfaces, and a description of each:

Object type	COM interface and description
Interface	IDirect3D
	COM interface object
Device	IDirect3DDevice
	Hardware device
Texture	IDirect3DTexture
	DirectDraw surface containing an image
Material	IDirect3DMaterial
	Surface properties, such as color and texture
Light	IDirect3DLight
	Light source
Viewport	IDirect3DViewport
	Screen region to draw to
Matrix	IDirect3DDevice
	4-by-4 homogeneous transformation matrix
ExecuteBuffer	IDirect3DExecuteBuffer
	List of vertex data and instructions about how to render it

Rendering is done by using execute buffers. These buffers contain vertex data and a list of opcodes that, when interpreted, instruct the rendering engine to produce an image. The execute buffer COM object contains only pointers and some descriptions of the format of the buffer. The contents of the buffer are dynamically allocated and can reside in the memory of the graphics card.

Each of the objects can effectively exist in one or more of the following forms:

- A COM object.
- A structure exposed to the developer that effectively contains a copy of the data in the COM object. This form is typically used to copy data into or out of the actual COM object.
- A handle. In this case, the data resides on the hardware and can be manipulated by it.

The following table shows the forms in which each of the Direct3D objects can exist:

	COM interface	Structure	Handle
Device	x		
Texture	x	x	x
Material	x	x	x
Light	x	x	
Viewport	x		
Matrix		x	x
ExecuteBuffer	x	x	

Direct3D Object Types

This section describes Direct3D object types. An application creates Direct3D objects roughly in the following order:

Direct3D objects
Device objects
Texture objects
Material objects
Light objects
Viewport objects
Execute-buffer objects

Direct3D Interface Objects

You can create a Direct3D interface object by calling the IDirectDraw::QueryInterface method, as follows:

```
lpDirectDraw->QueryInterface(
    IID_IDirect3D,  // IDirect3D interface ID
    lpD3D);         // Address of a Direct3D object
```

The object referred to by an IDirect3D interface contains a list of viewports, lights, materials, and devices. You can use the methods of the **IDirect3D** interface to create other objects or to find Direct3D devices.

Direct3D Device Objects

You can create a Direct3D device object by calling the IDirectDrawSurface::QueryInterface method for a backbuffer surface. The following example calls the IDirectDraw::CreateSurface and IDirectDrawSurface::GetAttachedSurface methods to retrieve the backbuffer surface.

```
lpDirectDraw->CreateSurface(
    lpDDSurfDesc,   // Address of a DDSURFACEDESC structure
    lpFrontBuffer,  // Address of a DIRECTDRAWSURFACE structure
    pUnkOuter);     // NULL
lpFrontBuffer->GetAttachedSurface(
    &ddscaps,         // Address of a DDSCAPS structure
    &lpBackBuffer);   // Address of a DIRECTDRAWSURFACE structure
lpBackBuffer->QueryInterface(
    GUIDforID3DDevice,   // ID for IDirect3DDevice interface
    lpD3DDevice);        // Address of a DIRECT3DDEVICE object
```

The first parameter of the call to the **IDirectDrawSurface::QueryInterface** method for the back buffer is the globally unique identifier (GUID) for the IDirect3DDevice interface. You can retrieve this GUID by calling the IDirect3D::EnumDevices method; the system supplies the GUID when it calls the D3DENUMDEVICESCALLBACK callback function you supply in the call to **IDirect3D::EnumDevices**.

A Direct3D device object resides on (or "is owned by") the interface list and has its own list of execute buffers and viewports. It also has a list of textures and materials, each of which points both to the next texture or material in the list and back to the device. For more information about this hierarchy, see Object Connectivity.

The methods of the **IDirect3DDevice** interface report hardware capabilities, maintain a list of viewports, manipulate matrix objects, and execute execute-buffer objects.

Matrices appear to you only as handles. You can create a Direct3D matrix by calling the IDirect3DDevice::CreateMatrix method, and you can set the contents of the matrix by calling the IDirect3DDevice::SetMatrix method. Matrix handles are used in execute buffers.

Direct3D Texture Objects

A texture is a rectangular array of colored pixels. (The rectangle does not necessarily have to be square, although the system deals most efficiently with square textures.) You can use textures for texture-mapping faces, in which case their dimensions must be powers of two. If your application uses the RGB color model, you can use 8-, 24-, and 32-bit textures. If you use the monochromatic (or ramp) color model, however, you can use only 8-bit textures.

You can retrieve an interface to a Direct3D texture object by calling the IDirectDrawSurface::QueryInterface method and specifying IID_IDirect3DTexture. An IDirect3DTexture interface is actually an interface to a DirectDrawSurface object, not to a distinct Direct3D texture object. For more information about the relationship between textures in Direct3D and surfaces in DirectDraw, see Direct3D Texture Interface.

The following example demonstrates how to create an IDirect3DTexture interface and then how to load the texture by calling the IDirect3DTexture::GetHandle and IDirect3DTexture::Load methods.

```
lpDDS->QueryInterface(IID_IDirect3DTexture,
    lpD3DTexture);  // Address of a DIRECT3DTEXTURE object
lpD3DTexture->GetHandle(
    lpD3DDevice,    // Address of a DIRECT3DDEVICE object
    lphTexture);    // Address of a D3DTEXTUREHANDLE
lpD3DTexture->Load(
    lpD3DTexture);  // Address of a DIRECT3DTEXTURE object
```

Texture objects reside on the interface list and point both to the next texture in a device's list and back to their associated device or devices. (For more information about this hierarchy, see Object Connectivity.) The texture handle is used in materials and execute buffers, and as a z-buffer for a viewport. You can use the **IDirect3DTexture** interface to load and unload textures, retrieve handles, and track changes to palettes.

Texture Wrapping

The texture coordinates of each face define the region in the texture that is mapped onto that particular face. Your application can use a wrap to calculate texture coordinates. For more information, see Direct3DRMWrap.

Your application can use the D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV render states (from the **D3DRENDERSTATETYPE** enumerated type) to specify how the rasterizer should interpret texture coordinates. The rasterizer always interpolates the shortest distance between texture coordinates; that is, a line. The path taken by this line, and the valid values for the u- and v-coordinates, varies with the use of the wrapping flags. If either or both flags is set, the line can wrap around the texture edge in the u or v direction, as if the texture had a cylindrical or toroidal topology.

- In flat wrapping mode, in which neither of the wrapping flags is set, the plane specified by the u- and v-coordinates is an infinite tiling of the texture. In this case, values greater than 1.0 are valid for u and v. The shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0.5, 0.5).
- If either D3DRENDERSTATE_WRAPU or D3DRENDERSTATE_WRAPV is set, the texture is an infinite cylinder with a circumference of 1.0. Texture coordinates greater than 1.0 are valid only in the dimension that is not wrapped. The shortest distance between texture coordinates varies with the wrapping flag; if D3DRENDERSTATE_WRAPU is set, the shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0, 0.5).
- If both D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV are set, the texture is a torus. Because the system is closed, texture coordinates greater that 1.0 are invalid. The shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0, 0).

Although texture coordinates that are outside the valid range may be truncated to valid values, this behavior is not defined.

Typically, applications set a wrap flag for cylindrical wraps when the intersection of the texture edges does not match the edges of the face, and do not set a wrap flag when more than half of a texture is applied to a single face.

For more information about wrapping, see Wrapping Types in the introduction to Direct3D Retained-

Mode objects.

Texture Filtering and Blending

After a texture has been mapped to a surface, the texture elements (texels) of the texture rarely correspond to individual pixels in the final image. A pixel in the final image can correspond to a large collection of texels or to a small piece of a single texel. You can use texture filtering to specify how to interpolate texel values to pixels.

You can use the D3DRENDERSTATE_TEXTUREMAG and D3DRENDERSTATE_TEXTUREMIN render states (from the **D3DRENDERSTATETYPE** enumerated type) to specify the type of texture filtering to use.

The D3DRENDERSTATE_TEXTUREMAPBLEND render state allows you to specify the type of texture blending. Texture blending combines the colors of the texture with the color of the surface to which the texture is being applied. This can be an effective way to achieve a translucent appearance. Texture blending can produce unexpected colors; the best way to avoid this is to ensure that the color of the material is white. The texture-blending options are specified in the D3DTEXTUREBLEND enumerated type.

You can use the D3DRENDERSTATE_SRCBLEND and D3DRENDERSTATE_DSTBLEND render states to specify how colors in the source and destination are combined. The combination options (called *blend factors*) are specified in the D3DBLEND enumerated type.

Mipmaps

A mipmap is a sequence of textures, each of which is a progressively lower resolution, prefiltered representation of the same image. Mipmapping is a computationally low-cost way of improving the quality of rendered textures. Each prefiltered image, or level, in the mipmap is a power of two smaller than the previous level.

You can use mipmaps when texture-filtering by specifying the appropriate filter mode in the D3DTEXTUREFILTER enumerated type. To find out what kinds of mipmapping support are provided by a device, use the flags specified in the **dwTextureFilterCaps** member of the D3DPRIMCAPS structure.

For more information about how to use DirectDraw to create mipmaps, see Mipmaps.

Transparency and Translucency

As already mentioned, one method for achieving the appearance of transparent or translucent textures is by using texture blending. You can also use alpha channels and the D3DRENDERSTATE_BLENDENABLE render state (from the **D3DRENDERSTATETYPE** enumerated type).

A more straightforward approach to achieving transparency or translucency is to use DirectDraw's support for *color keys*. Color keys are colors or ranges of colors that can be part of either the source or destination of a blit or overlay operation. You can specify whether these colors should always or never be overwritten.

For more information about DirectDraw's support for color keys, see Color Keying.

Direct3D Material Objects

You can create an interface to a Direct3D material by calling the IDirect3D::CreateMaterial method. The following example demonstrates how to create an IDirect3DMaterial interface. Then it demonstrates how to set the material and retrieve its handle by calling the IDirect3DMaterial::SetMaterial and IDirect3DMaterial::GetHandle methods.

```
lpDirect3D->CreateMaterial(
    lplpDirect3DMaterial,  // Address of a new material
    pUnkOuter);            // NULL
lpDirect3DMaterial->SetMaterial(
    lpD3DMat);             // Address of a D3DMATERIAL structure
lpDirect3DMaterial->GetHandle(
    lpD3DDevice,           // Address of a DIRECT3DDEVICE object
    lpD3DMat);             // Address of a D3DMATERIAL structure
```

Material objects reside on the interface list and point both to the next material in a device's list and back to their associated device or devices. (For more information about this hierarchy, see Object Connectivity.) A material contains colors and may contain a texture handle. The material handle is used inside execute buffers or to set the background of a viewport. You can use the **IDirect3DMaterial** interface to get and set materials, retrieve handles, and reserve colors.

Direct3D Light Objects

You can create a Direct3D light object by calling the IDirect3D::CreateLight method. The following example demonstrates how to create an IDirect3DLight interface, and then it sets the light by calling the IDirect3DLight::SetLight method.

```
lpDirect3D->CreateLight(
    lplpDirect3DLight,  // Address of a new light
    pUnkOuter);         // NULL
lpDirect3DLight->SetLight(
    lpLight);    // Address of a D3DLIGHT structure
```

Light objects reside on the interface list and on a viewport list. You can use the **IDirect3DLight** interface to get and set lights.

Direct3D Viewport Objects

You can create a Direct3D viewport object by calling the IDirect3D::CreateViewport method. The following example demonstrates how to create an IDirect3DViewport interface. Then it demonstrates how to add the viewport to a device by calling the IDirect3DDevice::AddViewport method and how to set up the viewport by calling the IDirect3DViewport::SetViewport, IDirect3DViewport::SetBackground, and IDirect3DViewport::AddLight methods.

```
lpDirect3D->CreateViewport(
    lplpDirect3DViewport,  // Address of a new viewport
    pUnkOuter);            // NULL
lpD3DDevice->AddViewport(
    lpD3DViewport)         // Attach viewport to device
lpD3DViewport->SetViewport(
    lpData);               // Address of a D3DVIEWPORT structure that
                           // Sets the viewport's location on the screen
lpD3DViewport->SetBackground(
    lphMat);               // Address of a D3DMATERIALHANDLE for
                           // background
lpD3DViewport->AddLight(
    lpD3DLight);           // Address of a light object
```

Viewport objects reside on the interface and device lists. The object maintains a list of lights as well as screen data, and it may have a material handle and a texture handle for the background. You can use the **IDirect3DViewport** interface to get and set backgrounds and viewports, add and delete lights, and transform vertices.

Direct3D Execute-Buffer Objects

Execute buffers contain a vertex list followed by an instruction stream. The instruction stream consists of operation codes, or *opcodes*, and the data that modifies those opcodes. For a description of execute buffers, see Execute Buffers.

You can create a Direct3D execute-buffer object by calling the IDirect3DDevice::CreateExecuteBuffer method.

```
lpD3DDevice->CreateExecuteBuffer(
    lpDesc,       // Address of a DIRECT3DEXECUTEBUFFERDESC structure
    lplpDirect3DExecuteBuffer,  // Address to contain a pointer to the
                                // Direct3DExecuteBuffer object
    pUnkOuter);   // NULL
```

Execute-buffer objects reside on a device list. You can use the **IDirect3DDevice::CreateExecuteBuffer** method to allocate space for the actual buffer, which may be on the hardware device.

The buffer is filled with two contiguous arrays of vertices and opcodes by using the following calls to the IDirect3DExecuteBuffer::Lock, IDirect3DExecuteBuffer::Unlock, and IDirect3DExecuteBuffer::SetExecuteData methods:

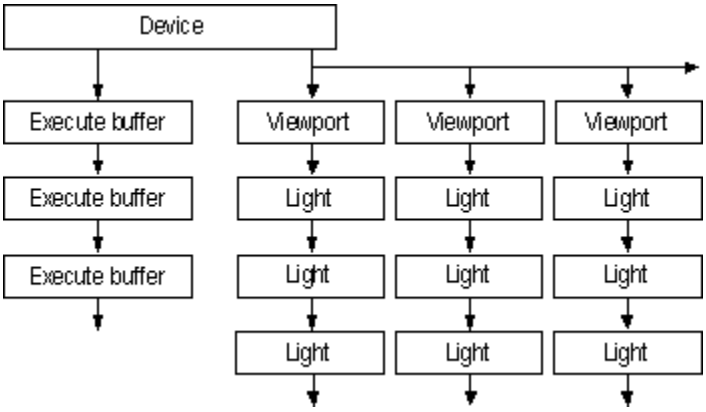
```
lpD3DExBuf->Lock(
    lpDesc);.      // Address of a DIRECT3DEXECUTEBUFFERDESC structure
//  .
//  .  Store contents through the supplied address
//  .
lpD3DExBuf->Unlock();
lpD3DExBuf->SetExecuteData(
    lpData);       // Address of a D3DEXECUTEDATA structure
```

The last call in the preceding example is to the **IDirect3DExecuteBuffer::SetExecuteData** method. This method notifies Direct3D where the two parts of the buffer reside relative to the address that was returned by the call to the **IDirect3DExecuteBuffer::Lock** method.

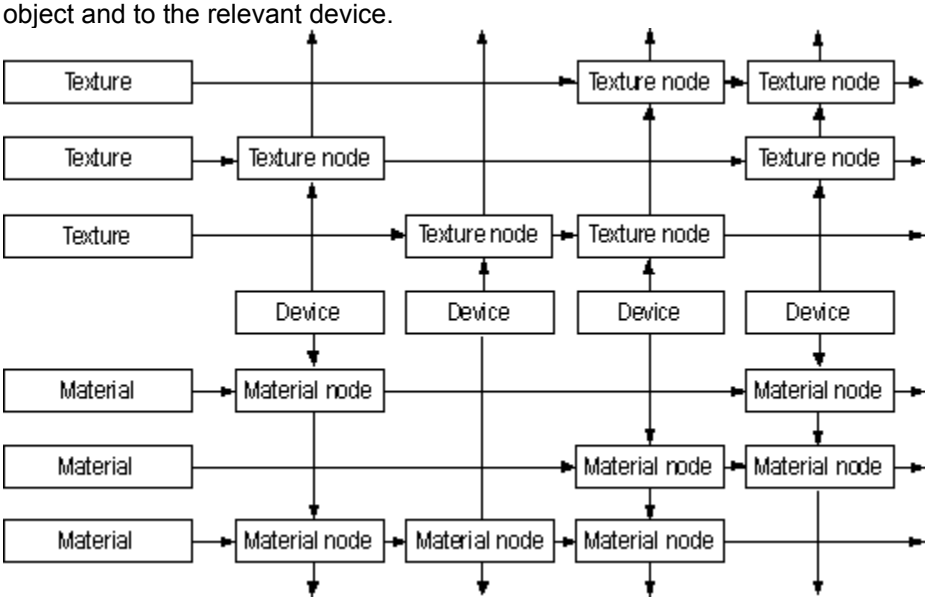
You can use the IDirect3DExecuteBuffer interface to get and set execute data, and to lock, unlock, optimize, and validate the execute buffer.

Object Connectivity

The following illustration shows how each execute buffer and viewport is owned by exactly one device, and each light is owned by exactly one viewport.



Material and texture objects can be associated with more than one device. Each node in the following illustration contains a pointer back to the head of the list it is in. (These pointers, however, are not shown in the illustration.) From any texture or material node, pointers lead back both to the texture or material object and to the relevant device.



You can create interface, device, and texture objects by calling the **QueryInterface** method. Material, light, and viewport objects are created by calls to the methods of the IDirect3D interface. Execute buffers and matrices are created by calls to the methods of the IDirect3DDevice interface. An interface object (Direct3D object) has a list of all created device, viewport, light, and material objects, but not execute buffers or textures.

Direct3D Immediate-Mode Tutorial

To create a Windows-based Direct3D Immediate-Mode application, you create DirectDraw and Direct3D objects, set render states, fill execute buffers, and execute those buffers. This section explains some simple Immediate-Mode sample code that is part of this SDK.

The D3dmain.cpp file in the samples provided with this SDK is used as a basis for all of the other Immediate-Mode samples. D3dmain.cpp contains the standard Windows framework of initialization, setting up a message loop, and creating a window procedure for message processing, but it also does some work that is specific to Direct3D Immediate-Mode applications. This work is discussed in the following sections:

The Immediate-Mode samples in this SDK include some code that is not documented here. In particular, this SDK includes a collection of helper functions collectively referred to as the D3DApp functions. You might find them useful when writing your own Immediate-Mode applications. These helper functions are referred to frequently in this documentation but are not covered exhaustively. They are implemented by the D3dapp.c, Ddcalls.c, D3dcalls.c, Texture.c, and Misc.c source files. The Stats.cpp source file sends frame-rate and screen-mode information to the screen.

Every sample that uses D3dmain.cpp must implement the following functions, which enable samples to customize their behavior:

- InitScene
- InitView
- RenderScene
- ReleaseView
- ReleaseScene
- OverrideDefaults

In addition, samples can use the SetMouseCallback and SetKeyboardCallback functions to read mouse and keyboard input.

Beginning Initialization

The first Direct3D task the WinMain function in D3dmain.cpp does is call the locally defined AppInit function, which creates the application window and initializes all objects needed to begin rendering. The WinMain function also implements the message pump for D3dmain.cpp and calls the locally defined RenderLoop and CleanUpAndPostQuit functions. The AppInit function calls other functions to help it do its work, and these functions, in turn, call still more functions; this group of initialization functions is the subject of most of this tutorial.

After performing some standard tasks in the initialization of a Windows application and initializing global variables to acceptable default settings, AppInit calls the InitScene function that each sample that uses D3dmain.cpp must implement. Simple sample applications, such as Oct1.c, implement versions of InitScene that do nothing but return TRUE. More complicated samples, such as Tunnel.c, use InitScene to allocate memory, generate points, and set global variables.

The last thing the AppInit function does before it returns is call the CreateD3DApp function, which is implemented in D3dmain.cpp. The functions called by CreateD3Dapp do much of the initialization work.

Creating DirectDraw and Direct3D Objects

The CreateD3DApp function in D3dmain.cpp is responsible for initializing the DirectDraw and Direct3D objects that are required before rendering begins. Important local functions called by CreateD3DApp include D3DAppCreateFromHWND, D3DAppGetRenderState, OverrideDefaults, D3DAppSetRenderState, ReleaseView, and InitView. Functions whose names begin with D3DApp are part of the D3DApp series of helper functions.

The same command-line options passed to the WinMain function are also passed to CreateD3DApp. Valid options are **-systemmemory** and **-emulation**. The **-systemmemory** option is used purely for debugging. The **-emulation** option prevents the application from using DirectDraw or Direct3D hardware.

CreateD3DApp calls the D3DAppAddTexture function to generate a series of textures. Then the D3DAppAddTexture function creates a source texture surface and object in system memory. Then it creates a second, initially empty, texture surface in video memory if hardware is present. The source texture is loaded into the destination texture surface and then discarded. This two-stage process allows a device to compress or reformat a texture map as it enters video memory. The code uses the IDirectDrawSurface::QueryInterface method to retrieve an interface to an IDirect3DTexture interface and the IDirect3DTexture::Load method to load the textures. The IDirect3DTexture::GetHandle method is used to create a list of texture handles.

After creating a list of textures, CreateD3DApp creates the DirectDraw and Direct3D objects required to start rendering. The code uses the D3DAppCreateFromHWND helper function. D3DAppCreateFromHWND uses functions that are implemented in the D3dapp.c, D3dcalls.c, Texture.c, and Ddcalls.c source files.

First, D3DAppCreateFromHWND uses the DirectDrawEnumerate and DirectDrawCreate functions to create and initialize the DirectDraw object, sets the values of global variables, and enumerates the display modes by calling the IDirectDraw2::EnumDisplayModes method.

D3DAppCreateFromHWND then creates the Direct3D object and enumerates the Direct3D device drivers. To create the Direct3D object, it calls the IDirectDraw::QueryInterface method, passing the IID_IDirect3D interface identifier. It uses IDirect3D::EnumDevices to enumerate the device drivers.

Calling **IDirect3D::EnumDevices** is not the easiest or even the best way to find Direct3D device drivers, however. Instead of setting up an enumeration routine, most Immediate-Mode applications use the IDirect3D::FindDevice method. This method allows you to simply specify the capabilities of the device you prefer—the system examines the available drivers and returns the globally unique identifier (GUID) for the first matching device. The system always searches the hardware first, so if both a hardware and a software device meet the required capabilities, the system returns the GUID for the hardware device.

After choosing a device driver and display mode (full-screen versus windowed), D3DAppCreateFromHWND creates front and back buffers for the chosen display mode. The code performs different actions based on whether the application is running in a window or the full screen, and whether video memory or system memory is being employed. If the application is running in a window, the code calls the IDirectDraw::CreateClipper method to create a clipper object and then calls the IDirectDrawClipper::SetHWnd method to attach it to the window and the IDirectDrawSurface::SetClipper method to attach it to the front buffer.

Then, D3DAppCreateFromHWND checks whether the front buffer is palettized. If it is, the code initializes the palette. First it creates the palette by calling the IDirectDraw::CreatePalette method, then it uses the IDirectDrawSurface::SetPalette method to set this as the palette for the front and back surfaces.

At this point the code calls the IDirectDraw::CreateSurface method to create a z-buffer, the IDirectDrawSurface::AddAttachedSurface method to attach the z-buffer to the back buffer, and the IDirectDrawSurface::GetSurfaceDesc method to determine whether the z-buffer is in video memory.

Then the code creates an IDirect3DDevice interface and uses it to enumerate the texture formats. The sample calls the IDirectDrawSurface::QueryInterface method to create the interface and the IDirect3DDevice::EnumTextureFormats method to enumerate the texture formats. When the textures have been enumerated, the code uses the same series of calls it employed in CreateD3DApp to load the textures and create a list of texture handles.

After using the driver's bit-depth and total video memory to filter the appropriate display modes, the code sets up the device-creation callback function, which is described in Setting Up the Device-Creation Callback Function.

When the device-creation callback function has been set up, D3DAppCreateFromHWND sets the application's render state, which is described in Setting the Immediate-Mode Render State.

When the D3DAppCreateFromHWND function has created the required Direct3D objects and set up the render state, it is almost finished. The function calls a local function to set the dirty rectangles for the front and back buffers to the entire client area, sets flags indicating that the application is initialized and that rendering can proceed, and returns TRUE.

The last part of the D3DAppCreateFromHWND function is its error-handling section. Whenever a call fails, the error-handling code jumps to the exit_with_error label. This section calls the callback function that destroys the device, resets the display mode and cooperative level if the application was running in full-screen mode, releases all the Direct3D and DirectDraw objects that were created, and returns FALSE.

Setting Up the Device-Creation Callback Function

The third parameter of the D3DAppCreateFromHWND function is an address of a callback function that is implemented as the AfterDeviceCreated function in D3dmain.cpp. AfterDeviceCreated creates the Direct3D viewport and returns it to D3DAppCreateFromHWND.

First the code calls the IDirect3D::CreateViewport method to create a viewport, and then it calls the IDirect3DDevice::AddViewport method to add the viewport to the newly created Direct3D device. After initializing the viewport's dimensions in a D3DVIEWPORT structure, the code calls the IDirect3DViewport::SetViewport method to set the viewport to those dimensions.

Then, the AfterDeviceCreated function calls the InitView function. InitView, like the InitScene function called earlier by D3dmain.cpp, must be implemented by each sample that uses D3dmain.cpp. One implementation of InitView is described in Initializing the Viewport.

After calling InitView and changing some of the menu items, AfterDeviceCreated finishes by calling the CleanUpAndPostQuit function. This function is described in Cleaning Up.

Initializing the Viewport

Each of the code samples that uses D3dmain.cpp must implement a version of the InitView function to set up the viewport and create the sample's execute buffers. This section discusses the implementation of InitView found in the Oct1.c sample.

First, InitView creates and initializes some materials, material handles, and texture handles. It uses the IDirect3D::CreateMaterial method to create a material, the IDirect3DMaterial::SetMaterial method to set the material data that InitView has just initialized, and the IDirect3DMaterial::GetHandle and IDirect3DViewport::SetBackground methods to set this material as the background for the viewport.

Now the InitView function sets the view, world, and projection matrices for the viewport. InitView uses the MAKE_MATRIX macro to create and set matrices. MAKE_MATRIX is defined in D3dmacs.h as follows:

```
#define MAKE_MATRIX(lpDev, handle, data) \
    if (lpDev->lpVtbl->CreateMatrix(lpDev, &handle) != D3D_OK) \
        return FALSE; \
    if (lpDev->lpVtbl->SetMatrix(lpDev, handle, &data) != D3D_OK) \
        return FALSE
```

As you can see, MAKE_MATRIX is simply a convenient way of calling the IDirect3DDevice::CreateMatrix and IDirect3DDevice::SetMatrix methods in a single step.

Now InitView creates and sets up an execute buffer. After initializing the members of a D3DEXECUTEBUFFERDESC structure, the code calls the IDirect3DDevice::CreateExecuteBuffer method to create the execute buffer and IDirect3DExecuteBuffer::Lock to lock it so that it can be filled.

InitView fills the execute buffer by using the OP_STATE_TRANSFORM and STATE_DATA macros from D3dmacs.h. These macros are described in Setting the Immediate-Mode Render State, along with more information about working with execute buffers.

When the execute buffer has been set up, InitView calls the IDirect3DExecuteBuffer::Unlock method to unlock it, the IDirect3DExecuteBuffer::SetExecuteData method to set the data into the buffer, and then the IDirect3DDevice::BeginScene, IDirect3DDevice::Execute, and IDirect3DDevice::EndScene methods to execute the execute buffer. Because the function has no further use for this execute buffer, it then calls IDirect3DExecuteBuffer::Release.

The InitView function now sets up two more materials by using the same procedure it used to set the materials earlier: it uses the IDirect3D::CreateMaterial method to create a material, the IDirect3DMaterial::SetMaterial method to set the material data (after filling the members of the D3DMATERIAL structure), and the IDirect3DMaterial::GetHandle method to retrieve a handle to the material. These handles are used later with the D3DLIGHTSTATE_MATERIAL member of the **D3DLIGHTSTATETYPE** enumerated type to associate lights with the new materials.

Now InitView sets up the vertices. The code uses the D3DVALP macro to convert floating-point numbers into the D3DVALUE members of the D3DVERTEX structure. It also uses the D3DRMVectorNormalize function to normalize the x-coordinate of the normal vector for each of the vertices it sets up.

When the vertices have been set up, InitView creates another execute buffer, copies the vertices to the buffer, and sets the execute data. It does not execute the execute buffer, however; this time, executing the execute buffer occurs in the rendering loop.

Finally, InitView sets up the lights for Oct1.c. After initializing the members of a D3DLIGHT structure, it calls the IDirect3D::CreateLight, IDirect3DLight::SetLight, and IDirect3DViewport::AddLight methods to add the light to the viewport.

Setting the Immediate-Mode Render State

The D3DAppISetRenderState function in the D3dcalls.c source file creates and executes an execute buffer that sets the render state and light state for the current viewport. The D3DAppCreateFromHWND function calls D3DAppISetRenderState from D3dapp.c; generally, the sample code calls D3DAppISetRenderState whenever the render state needs to be set or reset. This section reproduces the D3DAppISetRenderState function (except for some error-checking code).

After setting some local variables, including the D3DEXECUTEBUFFERDESC and D3DEXECUTEDATA structures, D3DAppISetRenderState calls the IDirect3DDevice::CreateExecuteBuffer method to create an execute buffer. When the execute buffer has been created, the code calls the IDirect3DExecuteBuffer::Lock method to lock it so that it can be filled.

```
BOOL D3DAppISetRenderState()
{
D3DEXECUTEBUFFERDESC debDesc;
D3DEXECUTEDATA d3dExData;
LPDIRECT3DEXECUTEBUFFER lpD3DExCmdBuf = NULL;
LPVOID lpBuffer, lpInsStart;
size_t size;

// Create an execute buffer of the required size and lock it
// so that it can be filled.

size = 0;
size += sizeof(D3DINSTRUCTION) * 3;
size += sizeof(D3DSTATE) * 17;
memset(&debDesc, 0, sizeof(D3DEXECUTEBUFFERDESC));
debDesc.dwSize = sizeof(D3DEXECUTEBUFFERDESC);
debDesc.dwFlags = D3DDEB_BUFSIZE;
debDesc.dwBufferSize = size;

LastError = d3dappi.lpD3DDevice->lpVtbl->CreateExecuteBuffer(
    d3dappi.lpD3DDevice, &debDesc, &lpD3DExCmdBuf, NULL);

LastError = lpD3DExCmdBuf->lpVtbl->Lock(lpD3DExCmdBuf, &debDesc);
memset(debDesc.lpData, 0, size);

lpInsStart = debDesc.lpData;
lpBuffer = lpInsStart;
```

The *d3dappi.lpD3DDevice* parameter in the call to IDirect3DDevice::CreateExecuteBuffer is a pointer to a Direct3DDevice object. The **lpData** member of the *debDesc* variable (a D3DEXECUTEBUFFERDESC structure) is a pointer to the actual data in the execute buffer.

Now the D3DAppISetRenderState function sets the render states. It uses the OP_STATE_DATA macro to help do some of this work. This macro, in turn, uses the PUTD3DINSTRUCTION macro. These macros are defined as follows in the D3dmacs.h header file in this SDK:

```
#define PUTD3DINSTRUCTION(op, sz, cnt, ptr) \
    ((LPD3DINSTRUCTION) ptr)->bOpcode = op; \
    ((LPD3DINSTRUCTION) ptr)->bSize = sz; \
    ((LPD3DINSTRUCTION) ptr)->wCount = cnt; \
    ptr = (void *)(((LPD3DINSTRUCTION) ptr) + 1)
#define OP_STATE_RENDER(cnt, ptr) \
    PUTD3DINSTRUCTION(D3DOP_STATERENDER, sizeof(D3DSTATE), cnt, ptr)
```

Notice that the PUTD3DINSTRUCTION macro is little more than an initializer for the D3DINSTRUCTION structure. D3DOP_STATERENDER, the first parameter of the call to PUTD3DINSTRUCTION in the OP_STATE_RENDER macro, is one of the opcodes in the D3DOPCODE enumerated type; the second parameter is the size of the D3DSTATE structure.

The STATE_DATA macro, which is also defined in D3dmacs.h, handles the render states. To work with the render states, it uses pointers to members in the D3DSTATE structure—in particular, a pointer to the D3DRENDERSTATETYPE enumerated type.

```
#define STATE_DATA(type, arg, ptr) \
    ((LPD3DSTATE) ptr)->drstRenderStateType = (D3DRENDERSTATETYPE)type; \
    ((LPD3DSTATE) ptr)->dwArg[0] = arg; \
    ptr = (void *)(((LPD3DSTATE) ptr) + 1)
```

The following code fragment from D3DAppISetRenderState uses the OP_STATE_RENDER and STATE_DATA macros to set 14 render states. The d3dapprs structure is the D3DAppRenderState structure, which is defined in the D3dapp.h header file.

```
OP_STATE_RENDER(14, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_SHADEMODE, d3dapprs.ShadeMode, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREPERSPECTIVE,
        d3dapprs.bPerspCorrect, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZENABLE, d3dapprs.bZBufferOn &&
        d3dappi.ThisDriver.bDoesZBuffer, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZWRITEENABLE, d3dapprs.bZBufferOn,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZFUNC, D3DCMP_LESSEQUAL, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREMAG, d3dapprs.TextureFilter,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREMIN, d3dapprs.TextureFilter,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREMAPBLEND, d3dapprs.TextureBlend,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_FILLMODE, d3dapprs.FillMode, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_DITHERENABLE, d3dapprs.bDithering,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_SPECULARENABLE, d3dapprs.bSpecular,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ANTIALIAS, d3dapprs.bAntialiasing,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_FOGENABLE, d3dapprs.bFogEnabled,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_FOGCOLOR, d3dapprs.FogColor, lpBuffer);
```

Now the OP_STATE_RENDER and STATE_DATA macros set three light states. The OP_EXIT macro simply uses the PUTD3DINSTRUCTION macro to call the D3DOP_EXIT opcode from the **D3DOPCODE** enumerated type.

```
OP_STATE_LIGHT(3, lpBuffer);
    STATE_DATA(D3DLIGHTSTATE_FOGMODE, d3dapprs.bFogEnabled ?
        d3dapprs.FogMode : D3DFOG_NONE, lpBuffer);
    STATE_DATA(D3DLIGHTSTATE_FOGSTART,
        *(unsigned long*)&d3dapprs.FogStart, lpBuffer);
    STATE_DATA(D3DLIGHTSTATE_FOGEND, *(unsigned long*)&d3dapprs.FogEnd,
        lpBuffer);
OP_EXIT(lpBuffer);
```

Now that the render states have been set, D3DAppISetRenderState unlocks the execute buffer by calling the IDirect3DExecuteBuffer::Unlock method. It sets the execute data by calling the IDirect3DExecuteBuffer::SetExecuteData method. Finally, it begins the scene, executes the execute buffer, and ends the scene again by calling the IDirect3DDevice::BeginScene, IDirect3DDevice::Execute, and IDirect3DDevice::EndScene methods.

```
LastError = lpD3DExCmdBuf->lpVtbl->Unlock(lpD3DExCmdBuf);

memset(&d3dExData, 0, sizeof(D3DEXECUTEDATA));
```

```
d3dExData.dwSize = sizeof(D3DEXECUTEDATA);
d3dExData.dwInstructionOffset = (ULONG) 0;
d3dExData.dwInstructionLength = (ULONG) ((char*)lpBuffer -
    (char*)lpInsStart);
lpD3DExCmdBuf->lpVtbl->SetExecuteData(lpD3DExCmdBuf, &d3dExData);
LastError =
    d3dappi.lpD3DDevice->lpVtbl->BeginScene(d3dappi.lpD3DDevice);
LastError =
    d3dappi.lpD3DDevice->lpVtbl->Execute(d3dappi.lpD3DDevice,
        lpD3DExCmdBuf, d3dappi.lpD3DViewport);
LastError = d3dappi.lpD3DDevice->lpVtbl->EndScene(d3dappi.lpD3DDevice);
```

The D3DAppISetRenderState has finished its work with the execute buffer, so it calls the IDirect3DExecuteBuffer::Release method to release it, and it returns.

```
lpD3DExCmdBuf->lpVtbl->Release(lpD3DExCmdBuf);
return TRUE;
}
```

Completing Initialization

The CreateD3DApp function that is called by the AppInit function in WinMain has created most of the underpinnings of a Direct3D application, but it hasn't finished yet. Before D3dmain.cpp calls the rendering loop, CreateD3DApp must complete a few more tasks.

After copying the current render states into the application's local D3DAppRenderState structure, CreateD3DApp calls the OverrideDefaults function. OverrideDefaults is another function that must be supported by all samples that use D3dmain.cpp. Sometimes an application will do very little in its OverrideDefaults function; for example, the Oct1.c sample simply replaces the default name with the string, "Octagon D3D Example".

Now CreateD3DApp calls the D3DAppSetRenderState function, which simply checks the status of the saved render states and either resets them (if no render states were provided in the call) or saves them before calling the D3DAppISetRenderState function, which is discussed in Setting the Immediate-Mode Render State.

As a final step before entering the rendering loop, CreateD3DApp calls the ReleaseView and InitView functions. These functions, like OverrideDefaults, are implemented by each sample application. ReleaseView simply releases any objects created in a previous call to the InitView function. (InitView, as you recall, is first called as part of the AfterDeviceCreated callback function.) This final call to InitView sets up the viewport again (in case there have been changes since the device was created) and re-creates the sample's execute buffers. For more information about InitView, see Initializing the Viewport.

This is the end not only of the CreateD3DApp function in D3dmain.cpp; it is also the end of the AppInit function. Now that the code has finished with the initialization part of the work, it can enter the rendering loop.

Running the Rendering Loop

After initializing the application, the WinMain function in D3dmain.cpp sets up the message loop. The code monitors the message queue until there are no pressing messages and then calls the RenderLoop function. RenderLoop is defined in D3dmain.cpp. WinMain maintains a count of the number of times RenderLoop can fail; if it fails more than three times, the application quits.

The RenderLoop function renders the next frame in the scene and updates the window.

The first task in the RenderLoop function is to restore any lost DirectDraw surfaces, if possible. It calls the IDirectDrawSurface::IsLost method to identify lost surfaces and IDirectDrawSurface::Restore to restore them. Then RenderLoop clears the back buffer and, if enabled, the z-buffer, by calling the IDirect3DRMViewport::Clear method.

Now RenderLoop calls the RenderScene function implemented by the sample application calling D3dmain.cpp. A sample's RenderScene function can take into account a complex series of special cases, or it can be as simple as the implementation in Oct1.c. Then, the RenderScene function executes the execute buffer by calling IDirect3DDevice::BeginScene, IDirect3DDevice::Execute, and IDirect3DDevice::EndScene, retrieves any new execute data by calling IDirect3DExecuteBuffer::GetExecuteData, updates the screen extents, and then calls a local TickScene function that changes the viewing position by calling the IDirect3DDevice::SetMatrix method.

Finally, the RenderLoop function calls the D3DAppRenderExtents helper function to keep track of the changed sections of the front and back buffers and blits or flips the back buffer to the front buffer.

Cleaning Up

Whenever an error occurs from which the application cannot recover, or the application receives a WM_QUIT or MENU_EXIT message, the application calls the CleanUpAndPostQuit function. CleanUpAndPostQuit does some simple error-checking and then calls the ReleaseScene function. ReleaseScene is the last of the functions that must be implemented by sample applications using D3dmain.cpp. This is an application's chance to release any remaining objects or free memory. For simple applications, like Oct1.c, ReleaseScene is simply a stub.

Finally, the CleanUpAndPostQuit function calls the **PostQuitMessage** function to end the application.

Direct3DRMCreate

```
HRESULT Direct3DRMCreate(LPDIRECT3DRM FAR * lplpD3DRM);
```

Creates an instance of a Direct3DRM object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRM
Address of a pointer that will be initialized with a valid Direct3DRM pointer if the call succeeds.

D3DRMColorGetAlpha

```
D3DVALUE D3DRMColorGetAlpha(D3DCOLOR d3drmc);
```

Retrieves the alpha component of a color.

- Returns the alpha value if successful, or zero otherwise.

d3drmc
Color from which the alpha component is retrieved.

D3DRMColorGetBlue

```
D3DVALUE D3DRMColorGetBlue(D3DCOLOR d3drmc);
```

Retrieves the blue component of a color.

- Returns the blue value if successful, or zero otherwise.

d3drmc
Color from which the blue component is retrieved.

D3DRMColorGetGreen

```
D3DVALUE D3DRMColorGetGreen(D3DCOLOR d3drmc);
```

Retrieves the green component of a color.

- Returns the green value if successful, or zero otherwise.

d3drmc
Color from which the green component is retrieved.

D3DRMColorGetRed

```
D3DVALUE D3DRMColorGetRed(D3DCOLOR d3drmc);
```

Retrieves the red component of a color.

- Returns the red value if successful, or zero otherwise.

d3drmc
Color from which the red component is retrieved.

D3DRMCreateColorRGB

```
D3DVALUE D3DRMCreateColorRGB(D3DVALUE red, D3DVALUE green,
    D3DVALUE blue);
```

Creates an RGB color from supplied red, green, and blue components.

- Returns the new RGB value if successful, or zero otherwise.

red, *green*, and *blue*
Components of the RGB color.

D3DRMCreateColorRGBA

```
D3DVALUE D3DRMCreateColorRGBA(D3DVALUE red, D3DVALUE green,
    D3DVALUE blue, D3DVALUE alpha);
```

Creates an RGBA color from supplied red, green, blue, and alpha components.

- Returns the new RGBA value if successful, or zero otherwise.

red, *green*, *blue*, and *alpha*
Components of the RGBA color.

D3DRMFREEFUNCTION

```
typedef VOID (*D3DRMFREEFUNCTION)(LPVOID lpArg);
typedef D3DRMFREEFUNCTION *LPD3DRMFREEFUNCTION;
```

Frees memory. This function is application-defined.

- No return value.

lpArg
Address of application-defined data.

Applications might define their own memory-freeing function if the standard C run-time routines do not meet their requirements.

D3DRMMALLOCFUNCTION

```
typedef LPVOID (*D3DRMMALLOCFUNCTION)(DWORD dwSize);
typedef D3DRMMALLOCFUNCTION *LPD3DRMMALLOCFUNCTION;
```

Allocates memory. This function is application-defined.

- Returns the address of the allocated memory if successful, or zero otherwise.

dwSize
Specifies the size, in bytes, of the memory that will be allocated.

Applications might define their own memory-allocation function if the standard C run-time routines do not meet their requirements.

D3DRMMatrixFromQuaternion

```
void D3DRMMatrixFromQuaternion (D3DRMMATRIX4D mat,
   LPD3DRMQUATERNION lpquat);
```

Calculates the matrix for the rotation that a unit quaternion represents.

- No return value.

mat
Address that will contain the calculated matrix when the function returns. (The D3DRMMATRIX4D type is an array.)

lpquat
Address of the D3DRMQUATERNION structure.

D3DRMQuaternionFromRotation

```
LPD3DRMQUATERNION D3DRMQuaternionFromRotation(LPD3DRMQUATERNION lpquat,
    LPD3DVECTOR lpv, D3DVALUE theta);
```

Retrieves a unit quaternion that represents a rotation of a specified number of radians around the given axis.

- Returns the address of the unit quaternion that was passed as the first parameter if successful, or zero otherwise.

lpquat
Address of a D3DRMQUATERNION structure that will contain the result of the operation.

lpv
Address of a D3DVECTOR structure specifying the axis of rotation.

theta
Number of radians to rotate around the axis specified by the *lpv* parameter.

D3DRMQuaternionMultiply

```
LPD3DRMQUATERNION D3DRMQuaternionMultiply(LPD3DRMQUATERNION lpq,
    LPD3DRMQUATERNION lpa, LPD3DRMQUATERNION lpb);
```

Calculates the product of two quaternion structures.

- Returns the address of the quaternion that was passed as the first parameter if successful, or zero otherwise.

lpq
Address of the D3DRMQUATERNION structure that will contain the product of the multiplication.

lpa and *lpb*
Addresses of the **D3DRMQUATERNION** structures that will be multiplied together.

D3DRMQuaternionSlerp

```
LPD3DRMQUATERNION D3DRMQuaternionSlerp(LPD3DRMQUATERNION lpq,
    LPD3DRMQUATERNION lpa, LPD3DRMQUATERNION lpb, D3DVALUE alpha);
```

Interpolates between two quaternion structures, using spherical linear interpolation.

- Returns the address of the quaternion that was passed as the first parameter if successful, or zero otherwise.

lpq
Address of the D3DRMQUATERNION structure that will contain the interpolation.

lpa and *lpb*
Addresses of the **D3DRMQUATERNION** structures that are used as the starting and ending points for the interpolation, respectively.

alpha
Value between 0 and 1 that specifies how far to interpolate between *lpa* and *lpb*.

D3DRMREALLOCFUNCTION

```
typedef LPVOID (*D3DRMREALLOCFUNCTION)(LPVOID lpArg,
    DWORD dwSize);
typedef D3DRMREALLOCFUNCTION *LPD3DRMREALLOCFUNCTION;
```

Reallocates memory. This function is application-defined.

- Returns an address of the reallocated memory if successful, or zero otherwise.

lpArg
Address of application-defined data.

dwSize
Size, in bytes, of the reallocated memory.

Applications may define their own memory-reallocation function if the standard C run-time routines do not meet their requirements.

D3DRMVectorAdd

```
LPD3DVECTOR D3DRMVectorAdd(LPD3DVECTOR lpd, LPD3DVECTOR lps1,
    LPD3DVECTOR lps2);
```

Adds two vectors.

- Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

lpd
Address of a D3DVECTOR structure that will contain the result of the addition.

lps1 and *lps2*
Addresses of the **D3DVECTOR** structures that will be added together.

D3DRMVectorCrossProduct

```
LPD3DVECTOR D3DRMVectorCrossProduct(LPD3DVECTOR lpd, LPD3DVECTOR lps1,
    LPD3DVECTOR lps2);
```

Calculates the cross product of the two vector arguments.

- Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

lpd
Address of a D3DVECTOR structure that will contain the result of the cross product.

lps1 and *lps2*
Addresses of the **D3DVECTOR** structures from which the cross product is produced.

D3DRMVectorDotProduct

```
D3DVALUE D3DRMVectorDotProduct(LPD3DVECTOR lps1, LPD3DVECTOR lps2);
```

Returns the vector dot product.

- Returns the result of the dot product if successful, or zero otherwise.

lps1 and *lps2*
Addresses of the D3DVECTOR structures from which the dot product is produced.

D3DRMVectorModulus

```
D3DVALUE D3DRMVectorModulus(LPD3DVECTOR lpv);
```

Returns the length of a vector (that is, sqrt(x*x + y*y + z*z)).

- Returns the length of the D3DVECTOR structure if successful, or zero otherwise.

lpv
Address of the **D3DVECTOR** structure whose length is returned.

D3DRMVectorNormalize

```
LPD3DVECTOR D3DRMVectorNormalize(LPD3DVECTOR lpv);
```

Scales a vector so that its modulus is 1.

- Returns the address of the vector that was passed as the first parameter if successful, or zero if an error occurs, such as if, for example, a zero vector was passed.

lpv
Address of a D3DVECTOR structure that will contain the result of the scaling operation.

D3DRMVectorRandom

```
LPD3DVECTOR D3DRMVectorRandom(LPD3DVECTOR lpd);
```

Returns a random unit vector.

- Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

lpd
Address of a D3DVECTOR structure that will contain a random unit vector.

D3DRMVectorReflect

```
LPD3DVECTOR D3DRMVectorReflect(LPD3DVECTOR lpd, LPD3DVECTOR lpRay,
    LPD3DVECTOR lpNorm);
```

Reflects a ray about a given normal.

- Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

lpd
Address of a D3DVECTOR structure that will contain the result of the operation.

lpRay
Address of a **D3DVECTOR** structure that will be reflected about a normal.

lpNorm
Address of a **D3DVECTOR** structure specifying the normal about which the vector specified in *lpRay* is reflected.

D3DRMVectorRotate

```
LPD3DVECTOR D3DRMVectorRotate(LPD3DVECTOR lpr, LPD3DVECTOR lpv,
    LPD3DVECTOR lpaxis, D3DVALUE theta);
```

Rotates a vector around a given axis.

- Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

lpr
Address of a D3DVECTOR structure that will contain the result of the operation.

lpv
Address of a **D3DVECTOR** structure that will be rotated around the given axis.

lpaxis
Address of a **D3DVECTOR** structure that is the axis of rotation.

theta
The rotation in radians.

D3DRMVectorScale

```
LPD3DVECTOR D3DRMVectorScale(LPD3DVECTOR lpd, LPD3DVECTOR lps,
    D3DVALUE factor);
```

Scales a vector uniformly in all three axes.

- Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

lpd
Address of a D3DVECTOR structure that will contain the result of the operation.

lps
Address of a **D3DVECTOR** structure that this function scales.

factor
Scaling factor. A value of 1 does not change the scaling; a value of 2 doubles it, and so on.

D3DRMVectorSubtract

```
LPD3DVECTOR D3DRMVectorSubtract(LPD3DVECTOR lpd, LPD3DVECTOR lps1,
    LPD3DVECTOR lps2);
```

Subtracts two vectors.

- Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

lpd
Address of a D3DVECTOR structure that will contain the result of the operation.

lps1
Address of the **D3DVECTOR** structure from which *lps2* is subtracted.

lps2
Address of the **D3DVECTOR** structure that is subtracted from *lps1*.

D3DRMDEVICEPALETTECALLBACK

```
void (*D3DRMDEVICEPALETTECALLBACK)
    (LPDIRECT3DRMDEVICE lpDirect3DRMDev, LPVOID lpArg, DWORD dwIndex,
    LONG red, LONG green, LONG blue);
```

Enumerates palette entries. This callback function is application-defined.

- No return value.

lpDirect3DRMDev
Address of the IDirect3DRMDevice interface for this device.

lpArg
Address of application-defined data passed to this callback function.

dwIndex
Index of the palette entry being described.

red, *green*, and *blue*
Red, green, and blue components of the color at the given index in the palette.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DRMFRAMEMOVECALLBACK

```
void (*D3DRMFRAMEMOVECALLBACK)(LPDIRECT3DRMFRAME lpD3DRMFrame,
    LPVOID lpArg, D3DVALUE delta);
```

Enables an application to apply customized algorithms when a frame is moved or updated. You can use this callback function to compensate for changing frame rates. This callback function is application-defined.

- No return value.

lpD3DRMFrame
Address of the Direct3DRMFrame object that is being moved.

lpArg
Address of application-defined data passed to this callback function.

delta
Amount of change to apply to the movement. There are two components to the change in position of a frame: linear and rotational. The change in each component is equal to *velocity_of_component* × *delta*. Although either or both of these velocities can be set relative to any frame, the system automatically converts them to velocities relative to the parent frame for the purpose of applying time deltas.

Your application can synthesize the acceleration of a frame relative to its parent frame. To do so, on each tick your application should set the velocity of the child frame relative to itself to (*a* units per tick) × 1 tick, where *a* is the required acceleration. This is equal to *a* × *delta* units per tick. Internally, *a* × *delta* units per tick relative to the child frame is converted to (*v* + (*a* × *delta*)) units per tick relative to the parent frame, where *v* is the current velocity of the child relative to the parent.

You can add and remove this callback function from your application by using the IDirect3DRMFrame::AddMoveCallback and IDirect3DRMFrame::DeleteMoveCallback methods.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DRMLOADCALLBACK

```
void (*D3DRMLOADCALLBACK) (LPDIRECT3DRMOBJECT lpObject, REFIID ObjectGuid,
    LPVOID lpArg);
```

Loads objects named in a call to the IDirect3DRM::Load method. This callback function is application-defined.

- No return value.

lpObject
Address of the Direct3DRMObject being loaded.

ObjectGuid
Globally unique identifier (GUID) of the object being loaded.

lpArg
Address of application-defined data passed to this callback function.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DRMLOADTEXTURECALLBACK

```
HRESULT (*D3DRMLOADTEXTURECALLBACK) (char *tex_name, void *lpArg,
    LPDIRECT3DRMTEXTURE * lpD3DRMTex);
```

Loads texture maps from a file or resource named in a call to one of the **Load** methods. This callback function is application-defined.

- Should return D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

tex_name
Address of a string containing the name of the texture.

lpArg
Address of application-specific data.

lpD3DRMTex
Address of the Direct3DRMTexture object.

Applications can use this callback function to implement support for textures that are not in the Windows bitmap (.bmp) or Portable Pixmap (.ppm) P6 format.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DRMOBJECTCALLBACK

```
void (*D3DRMOBJECTCALLBACK)(LPDIRECT3DRMOBJECT lpD3DRMobj,
    LPVOID lpArg);
```

Enumerates objects in response to a call to the IDirect3DRM::EnumerateObjects method. This callback function is application-defined.

- No return value.

lpD3DRMobj

Address of an IDirect3DRMObject interface for the object being enumerated. The application must call the **Release** method for each enumerated object.

lpArg

Address of application-defined data passed to this callback function.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DRMUPDATECALLBACK

```
void (*D3DRMUPDATECALLBACK)(LPDIRECT3DRMDEVICE lpobj, LPVOID lpArg,
    int x, LPD3DRECT d3dRectUpdate);
```

Alerts the application whenever the device changes. This callback function is application-defined.

- No return value.

lpobj
Address of the Direct3DRMDevice object to which this callback function applies.

lpArg
Address of application-defined data passed to this callback function.

x
Number of rectangles specified in the *d3dRectUpdate* parameter.

d3dRectUpdate
Array of one or more D3DRECT structures that describe the area to be updated. The coordinates are specified in device units.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DRMUSERVISUALCALLBACK

```
int (*D3DRMUSERVISUALCALLBACK)(LPDIRECT3DRMUSERVISUAL lpD3DRMUV,
    LPVOID lpArg, D3DRMUSERVISUALREASON lpD3DRMUVreason,
    LPDIRECT3DRMDEVICE lpD3DRMDev, LPDIRECT3DRMVIEWPORT lpD3DRMview);
```

Alerts an application that supplies user-visual objects that it should execute the execute buffer. This function is application-defined.

- Returns TRUE if the *lpD3DRMUVreason* parameter is D3DRMUSERVISUAL_CANSEE and the user-visual object is visible in the viewport. Returns FALSE otherwise. If the *lpD3DRMUVreason* parameter is D3DRMUSERVISUAL_RENDER, the return value is application-defined. It is always safe to return TRUE.

lpD3DRMUV

Address of the Direct3DRMUserVisual object.

lpArg

Address of application-defined data passed to this callback function.

lpD3DRMUVreason

One of the members of the D3DRMUSERVISUALREASON enumerated type:

D3DRMUSERVISUAL_CANSEE

The application should return TRUE if the user-visual object is visible in the viewport. In this case, the application uses the device specified in the *lpD3DRMview* parameter.

D3DRMUSERVISUAL_RENDER

The application should render the user-visual element. In this case, the application uses the device specified in the *lpD3DRMDev* parameter.

lpD3DRMDev

Address of a Direct3DRMDevice object used to render the Direct3DRMUserVisual object.

lpD3DRMview

Address of a Direct3DRMViewport object used to determine whether the Direct3DRMUserVisual object is visible.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DRMWRAPCALLBACK

```
void (*D3DRMWRAPCALLBACK)(LPD3DVECTOR lpD3DVector,
    int* lpU, int* lpV, LPD3DVECTOR lpD3DRMVA, LPD3DVECTOR lpD3DRMVB,
    LPVOID lpArg);
```

This callback function is not supported.

Introduction to Array Interfaces

The array interfaces make it possible for your application to group objects into arrays, making it simpler to apply operations to the entire group. The following array interfaces are available:

IDirect3DRMArray
IDirect3DRMDeviceArray
IDirect3DRMFaceArray
IDirect3DRMFrameArray
IDirect3DRMLightArray
IDirect3DRMPickedArray
IDirect3DRMViewportArray
IDirect3DRMVisualArray

IDirect3DRMArray Interface Method Groups

The **IDirect3DRMArray** interface organizes groups of objects. Applications typically use array objects that are subsidiary to this interface, rather than using this interface directly. This interface supports the following methods:

Information	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMArray object without affecting the functionality of the original interface.

IDirect3DRMArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMArray.

- Returns the new reference count of the object.

When the Direct3DRMArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMArray::GetSize

```
DWORD GetSize();
```

Retrieves the size, in objects, of the Direct3DRMArray object.

- Returns the size.

IDirect3DRMArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMArray::QueryInterface** method allows Direct3DRMArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMArray.

- Returns the new reference count of the object.

The Direct3DRMArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRMDeviceArray Interface Method Groups

Applications use the methods of the **IDirect3DRMDeviceArray** interface to organize device objects. The methods can be organized into the following groups:

Information	GetElement
	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMDeviceArray object without affecting the functionality of the original interface. In addition, the **IDirect3DRMDeviceArray** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMDeviceArray object is obtained by calling the IDirect3DRM::GetDevices method.

IDirect3DRMDeviceArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMDeviceArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMDeviceArray.

- Returns the new reference count of the object.

When the Direct3DRMDeviceArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMDeviceArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMDeviceArray::GetElement

```
HRESULT GetElement(DWORD index, LPDIRECT3DRMDEVICE * lplpD3DRMDevice);
```

Retrieves a specified element in a Direct3DRMDeviceArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Element in the array.

lplpD3DRMDevice
Address that will be filled with a pointer to an IDirect3DRMDevice interface.

IDirect3DRMDeviceArray::GetSize

```
DWORD GetSize();
```

Retrieves the number of elements contained in a Direct3DRMDeviceArray object.

- Returns the number of elements.

IDirect3DRMDeviceArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMDeviceArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMDeviceArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMDeviceArray::QueryInterface** method allows Direct3DRMDeviceArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMDeviceArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMDeviceArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMDeviceArray.

- Returns the new reference count of the object.

The Direct3DRMDeviceArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMDeviceArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRMFaceArray Interface Method Groups

Applications use the methods of the **IDirect3DRMFaceArray** interface to organize faces in a mesh. The methods can be organized into the following groups:

Information	GetElement
	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMFaceArray object without affecting the functionality of the original interface. In addition, the **IDirect3DRMFaceArray** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMFaceArray object is obtained by calling the IDirect3DRMMeshBuilder::GetFaces method.

IDirect3DRMFaceArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMFaceArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFaceArray.

- Returns the new reference count of the object.

When the Direct3DRMFaceArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMFaceArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMFaceArray::GetElement

```
HRESULT GetElement(DWORD index, LPDIRECT3DRMFACE * lplpD3DRMFace);
```

Retrieves a specified element in a Direct3DRMFaceArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Element in the array.

lplpD3DRMFace
Address that will be filled with a pointer to an IDirect3DRMFace interface.

IDirect3DRMFaceArray::GetSize

```
DWORD GetSize();
```

Retrieves the number of elements contained in a Direct3DRMFaceArray object.

- Returns the number of elements.

IDirect3DRMFaceArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMFaceArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMFaceArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMFaceArray::QueryInterface** method allows Direct3DRMFaceArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMFaceArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMFaceArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFaceArray.

- Returns the new reference count of the object.

The Direct3DRMFaceArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMFaceArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRMFrameArray Interface Method Groups

Applications use the methods of the **IDirect3DRMFrameArray** interface to organize frame objects. The methods can be organized into the following groups:

Information	GetElement
	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMFrameArray object without affecting the functionality of the original interface. In addition, the **IDirect3DRMFrameArray** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMFrameArray object is obtained by calling the IDirect3DRMPickedArray::GetPick or IDirect3DRMFrame::GetChildren method.

IDirect3DRMFrameArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMFrameArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFrameArray.

- Returns the new reference count of the object.

When the Direct3DRMFrameArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMFrameArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMFrameArray::GetElement

```
HRESULT GetElement(DWORD index, LPDIRECT3DRMFRAME * lplpD3DRMFrame);
```

Retrieves a specified element in a Direct3DRMFrameArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Element in the array.

lplpD3DRMFrame
Address that will be filled with a pointer to an IDirect3DRMFrame interface.

IDirect3DRMFrameArray::GetSize

```
DWORD GetSize();
```

Retrieves the number of elements contained in a Direct3DRMFrameArray object.

- Returns the number of elements.

IDirect3DRMFrameArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMFrameArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMFrameArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMFrameArray::QueryInterface** method allows Direct3DRMFrameArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMFrameArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMFrameArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFrameArray.

- Returns the new reference count of the object.

The Direct3DRMFrameArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMFrameArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRMLightArray Interface Method Groups

Applications use the methods of the **IDirect3DRMLightArray** interface to organize light objects. The methods can be organized into the following groups:

Information	GetElement
	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMLightArray object without affecting the functionality of the original interface. In addition, the **IDirect3DRMLightArray** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMLightArray object is obtained by calling the IDirect3DRMFrame::GetLights method.

IDirect3DRMLightArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMLightArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMLightArray.

- Returns the new reference count of the object.

When the Direct3DRMLightArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMLightArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMLightArray::GetElement

```
HRESULT GetElement(DWORD index, LPDIRECT3DRMLIGHT * lplpD3DRMLight);
```

Retrieves a specified element in a Direct3DRMLightArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Element in the array.

lplpD3DRMLight
Address that will be filled with a pointer to an IDirect3DRMLight interface.

IDirect3DRMLightArray::GetSize

```
DWORD GetSize();
```

Retrieves the number of elements contained in a Direct3DRMLightArray object.

- Returns the number of elements.

IDirect3DRMLightArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMLightArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMLightArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMLightArray::QueryInterface** method allows Direct3DRMLightArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMLightArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMLightArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMLightArray.

- Returns the new reference count of the object.

The Direct3DRMLightArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMLightArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRMPickedArray Interface Method Groups

Applications use the methods of the **IDirect3DRMPickedArray** interface to organize pick objects. The methods can be organized into the following groups:

Information	GetPick
	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMPickedArray object without affecting the functionality of the original interface. In addition, the **IDirect3DRMPickedArray** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMPickedArray object is obtained by calling the IDirect3DRMViewport::Pick method.

IDirect3DRMPickedArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMPickedArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMPickedArray.

- Returns the new reference count of the object.

When the Direct3DRMPickedArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMPickedArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMPickedArray::GetPick

```
HRESULT GetPick(DWORD index, LPDIRECT3DRMVISUAL * lplpVisual,
    LPDIRECT3DRMFRAMEARRAY * lplpFrameArray,
    LPD3DRMPICKDESC lpD3DRMPickDesc);
```

Retrieves the Direct3DRMVisual and Direct3DRMFrame objects intersected by the specified pick.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index

Index into the pick array identifying the pick for which information will be retrieved.

lplpVisual

Address that will contain a pointer to the Direct3DRMVisual object associated with the specified pick.

lplpFrameArray

Address that will contain a pointer to the Direct3DRMFrameArray object associated with the specified pick.

lpD3DRMPickDesc

Address of a D3DRMPICKDESC structure specifying the pick position and face and group identifiers of the objects being retrieved.

IDirect3DRMPickedArray::GetSize

```
DWORD GetSize();
```

Retrieves the number of elements contained in a Direct3DRMPickedArray object.

- Returns the number of elements.

IDirect3DRMPickedArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMPickedArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMPickedArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMPickedArray::QueryInterface** method allows Direct3DRMPickedArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMPickedArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMPickedArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMPickedArray.

- Returns the new reference count of the object.

The Direct3DRMPickedArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMPickedArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRMViewportArray Interface Method Groups

Applications use the methods of the **IDirect3DRMViewportArray** interface to organize viewport objects. The methods can be organized into the following groups:

Information	GetElement
	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMViewportArray object without affecting the functionality of the original interface. In addition, the **IDirect3DRMViewportArray** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMViewportArray object is obtained by calling the IDirect3DRM::CreateFrame method.

IDirect3DRMViewportArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMViewportArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMViewportArray.

- Returns the new reference count of the object.

When the Direct3DRMViewportArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMViewportArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMViewportArray::GetElement

```
HRESULT GetElement(DWORD index, LPDIRECT3DRMVIEWPORT * lplpD3DRMViewport);
```

Retrieves a specified element in a Direct3DRMViewportArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Element in the array.

lplpD3DRMViewport
Address that will be filled with a pointer to an IDirect3DRMViewport interface.

IDirect3DRMViewportArray::GetSize

```
DWORD GetSize();
```

Retrieves the number of elements contained in a Direct3DRMViewportArray object.

- Returns the number of elements.

IDirect3DRMViewportArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMViewportArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMViewportArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMViewportArray::QueryInterface** method allows Direct3DRMViewportArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMViewportArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMViewportArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMViewportArray.

- Returns the new reference count of the object.

The Direct3DRMViewportArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMViewportArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRMVisualArray Interface Method Groups

Applications use the methods of the **IDirect3DRMVisualArray** interface to organize groups of visual objects. The methods can be organized into the following groups:

Information	GetElement
	GetSize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMVisualArray object without affecting the functionality of the original interface. In addition, the **IDirect3DRMVisualArray** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMVisualArray object is obtained by calling the IDirect3DRMFrame::GetVisuals method.

IDirect3DRMVisualArray::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMVisualArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMVisualArray.

- Returns the new reference count of the object.

When the Direct3DRMVisualArray object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMVisualArray::Release method to decrease the reference count of the object by 1.

IDirect3DRMVisualArray::GetElement

```
HRESULT GetElement(DWORD index, LPDIRECT3DRMVISUAL * lplpD3DRMVisual);
```

Retrieves a specified element in a Direct3DRMVisualArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Element in the array.

lplpD3DRMVisual
Address that will be filled with a pointer to an **IDirect3DRMVisual** interface.

IDirect3DRMVisualArray::GetSize

```
DWORD GetSize();
```

Retrieves the number of elements contained in a Direct3DRMVisualArray object.

- Returns the number of elements.

IDirect3DRMVisualArray::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMVisualArray object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMVisualArray.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMVisualArray::QueryInterface** method allows Direct3DRMVisualArray objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMVisualArray::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMVisualArray object by 1. This method is part of the IUnknown interface inherited by Direct3DRMVisualArray.

- Returns the new reference count of the object.

The Direct3DRMVisualArray object deallocates itself when its reference count reaches 0. Use the IDirect3DRMVisualArray::AddRef method to increase the reference count of the object by 1.

IDirect3DRM Interface Method Groups

Applications use the methods of the **IDirect3DRM** interface to create Direct3DRM objects and work with system-level variables. The methods can be organized into the following groups:

Animation	CreateAnimation CreateAnimationSet
Devices	CreateDevice CreateDeviceFromClipper CreateDeviceFromD3D CreateDeviceFromSurface GetDevices
Enumeration	EnumerateObjects
Faces	CreateFace
Frames	CreateFrame
IUnknown	AddRef QueryInterface Release
Lights	CreateLight CreateLightRGB
Materials	CreateMaterial
Meshes	CreateMesh CreateMeshBuilder
Miscellaneous	CreateObject CreateUserVisual GetNamedObject Load Tick
Search paths	AddSearchPath GetSearchPath SetSearchPath
Shadows	CreateShadow
Textures	CreateTexture CreateTextureFromSurface LoadTexture LoadTextureFromResource SetDefaultTextureColors

	SetDefaultTextureShades
Viewports	CreateViewport
Wraps	CreateWrap

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRM object without affecting the functionality of the original interface.

The **IDirect3DRM** COM interface is created by calling the Direct3DRMCreate function.

IDirect3DRM::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRM object by 1. This method is part of the IUnknown interface inherited by Direct3DRM.

- Returns the new reference count of the object.

When the Direct3DRM object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRM::Release method to decrease the reference count of the object by 1.

IDirect3DRM::AddSearchPath

```
HRESULT AddSearchPath(LPCSTR lpPath);
```

Adds a list of directories to the end of the current file search path.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpPath

Address of a null-terminated string specifying the path to add to the current search path.

For Windows, the path should be a list of directories separated by semicolons (;).

IDirect3DRM::CreateAnimation

```
HRESULT CreateAnimation(LPDIRECT3DRMANIMATION * lplpD3DRMAnimation);
```

Creates an empty Direct3DRMAnimation object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRMAnimation
Address that will be filled with a pointer to an IDirect3DRMAnimation interface if the call succeeds.

IDirect3DRM::CreateAnimationSet

```
HRESULT CreateAnimationSet (LPDIRECT3DRMANIMATIONSET *
lplpD3DRMAnimationSet);
```

Creates an empty Direct3DRMAnimationSet object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRMAnimationSet
Address that will be filled with a pointer to an IDirect3DRMAnimationSet interface if the call succeeds.

IDirect3DRM::CreateDevice

```
HRESULT CreateDevice(DWORD dwWidth, DWORD dwHeight,
    LPDIRECT3DRMDEVICE* lplpD3DRMDevice);
```

Not implemented on the Windows platform.

IDirect3DRM::CreateDeviceFromClipper

```
HRESULT CreateDeviceFromClipper(LPDIRECTDRAWCLIPPER lpDDClipper,
    LPGUID lpGUID, int width, int height,
    LPDIRECT3DRMDEVICE * lplpD3DRMDevice);
```

Creates a Direct3DRM Windows device by using a specified DirectDrawClipper object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpDDClipper
Address of a DirectDrawClipper object.

lpGUID
Address of a globally unique identifier (GUID). This parameter can be NULL.

width and *height*
Width and height of the device to be created.

lplpD3DRMDevice
Address that will be filled with a pointer to an IDirect3DRMDevice interface if the call succeeds.

If the *lpGUID* parameter is NULL, the system searches for a device with a default set of device capabilities. This is the recommended way to create a Retained-Mode device because it always works, even if the user installs new hardware.

The system describes the default settings by using the following flags from the D3DPRIMCAPS structure in internal device-enumeration calls:

D3DPCMPCAPS_LESSEQUAL
D3DPMISCCAPS_CULLCCW
D3DPRASTERCAPS_FOGVERTEX
D3DPSHADECAPS_ALPHAFLATSTIPPLED
D3DPTADDRESSCAPS_WRAP
D3DPTBLENDCAPS_COPY | D3DPTBLENDCAPS_MODULATE
D3DPTEXTURECAPS_PERSPECTIVE | D3DPTEXTURECAPS_TRANSPARENCY
D3DPTFILTERCAPS_NEAREST

If a hardware device is not found, the monochromatic (ramp) software driver is loaded. An application should enumerate devices instead of specifying NULL for *lpGUID* if it has special needs that are not met by this list of default settings.

IDirect3DRM::CreateDeviceFromD3D

```
HRESULT CreateDeviceFromD3D(LPDIRECT3D lpD3D,
    LPDIRECT3DDEVICE lpD3DDev, LPDIRECT3DRMDEVICE * lplpD3DRMDevice);
```

Creates a Direct3DRM Windows device by using specified Direct3D objects.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3D
Address of an instance of Direct3D.

lpD3DDev
Address of a Direct3D device object.

lplpD3DRMDevice
Address that will be filled with a pointer to an IDirect3DRMDevice interface if the call succeeds.

IDirect3DRM::CreateDeviceFromSurface

```
HRESULT CreateDeviceFromSurface(LPGUID lpGUID, LPDIRECTDRAW lpDD,
    LPDIRECTDRAWSURFACE lpDDSBack,
    LPDIRECT3DRMDEVICE * lplpD3DRMDevice);
```

Creates a Windows device for rendering from the specified DirectDraw surfaces.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpGUID

Address of the globally unique identifier (GUID) used as the required device driver. If this parameter is NULL, the default device driver is used.

lpDD

Address of the DirectDraw object that is the source of the DirectDraw surface.

lpDDSBack

Address of the DirectDrawSurface object that represents the back buffer.

lplpD3DRMDevice

Address that will be filled with a pointer to an IDirect3DRMDevice interface if the call succeeds.

IDirect3DRM::CreateFace

```
HRESULT CreateFace(LPDIRECT3DRMFACE * lplpd3drmFace);
```

Creates an instance of the IDirect3DRMFace interface.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpd3drmFace
Address that will be filled with a pointer to an IDirect3DRMFace interface if the call succeeds.

IDirect3DRM::CreateFrame

```
HRESULT CreateFrame(LPDIRECT3DRMFRAME lpD3DRMFrame,
    LPDIRECT3DRMFRAME* lplpD3DRMFrame);
```

Creates a new child frame of the given parent frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMFrame

Address of a frame that is to be the parent of the new frame.

lplpD3DRMFrame

Address that will be filled with a pointer to an IDirect3DRMFrame interface if the call succeeds.

The child frame inherits the motion attributes of its parent. For example, if the parent is moving with a given velocity, the child frame will also move with that velocity. Furthermore, if the parent is set rotating, the child frame will rotate about the origin of the parent. Frames that have no parent are called scenes. To create a scene, specify NULL as the parent. An application can create a frame with no parent and then attach it to a parent frame later by using the IDirect3DRMFrame::AddChild method.

IDirect3DRM::CreateLight

```
HRESULT CreateLight(D3DRMLIGHTTYPE d3drmltLightType,
    D3DCOLOR cColor, LPDIRECT3DRMLIGHT* lplpD3DRMLight);
```

Creates a new light source with the given type and color.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmltLightType
One of the lighting types given in the D3DRMLIGHTTYPE enumerated type.

cColor
Color of the light.

lplpD3DRMLight
Address that will be filled with a pointer to an IDirect3DRMLight interface if the call succeeds.

IDirect3DRM::CreateLightRGB

```
HRESULT CreateLightRGB(D3DRMLIGHTTYPE ltLightType, D3DVALUE vRed,
    D3DVALUE vGreen, D3DVALUE vBlue, LPDIRECT3DRMLIGHT* lplpD3DRMLight);
```

Creates a new light source with the given type and color.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

ltLightType
One of the lighting types given in the D3DRMLIGHTTYPE enumerated type.

vRed, vGreen, and *vBlue*
Color of the light.

lplpD3DRMLight
Address that will be filled with a pointer to an IDirect3DRMLight interface if the call succeeds.

IDirect3DRM::CreateMaterial

```
HRESULT CreateMaterial(D3DVALUE vPower,
    LPDIRECT3DRMMATERIAL * lplpD3DRMMaterial);
```

Creates a material with the given specular properties.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

vPower

Sharpness of the reflected highlights, with a value of 5 giving a metallic look and higher values giving a more plastic look to the rendered surface.

lplpD3DRMMaterial

Address that will be filled with a pointer to an IDirect3DRMMaterial interface if the call succeeds.

IDirect3DRM::CreateMesh

```
HRESULT CreateMesh(LPDIRECT3DRMMESH* lplpD3DRMMesh);
```

Creates a new mesh object with no faces. The mesh is not visible until it is added to a frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRMMesh
Address that will be filled with a pointer to an IDirect3DRMMesh interface if the call succeeds.

IDirect3DRM::CreateMeshBuilder

```
HRESULT CreateMeshBuilder(LPDIRECT3DRMMESHBUILDER* lplpD3DRMMeshBuilder);
```

Creates a new mesh builder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRMMeshBuilder
Address that will be filled with a pointer to an IDirect3DRMMeshBuilder interface if the call succeeds.

IDirect3DRM::CreateObject

```
HRESULT CreateObject(REFCLSID rclsid, LPUNKNOWN pUnkOuter,
    REFIID riid, LPVOID FAR* ppv);
```

Creates a new object without initializing the object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rclsid
Class identifier for the new object.

pUnkOuter
Allows COM aggregation features.

riid
Interface identifier of the object to be created.

ppv
Address of a pointer to the object when the method returns.

An application that calls this method must initialize the object that has been created. (The other creation methods of the IDirect3DRM interface initialize the object automatically.) To initialize the new object, you should use the **Init** method for that object. An application should call the **Init** method only once to initialize any given object.

Applications can use this method to implement aggregation in Direct3DRM objects.

IDirect3DRM::CreateShadow

```
HRESULT CreateShadow(LPDIRECT3DRMVISUAL lpVisual,
    LPDIRECT3DRMLIGHT lpLight, D3DVALUE px, D3DVALUE py, D3DVALUE pz,
    D3DVALUE nx, D3DVALUE ny, D3DVALUE nz,
    LPDIRECT3DRMVISUAL * lplpShadow);
```

Creates a shadow by using the specified visual and light, projecting the shadow onto the specified plane. The shadow is a visual that should be added to the frame that contains the visual.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpVisual
Address of the Direct3DRMVisual object that is casting the shadow.

lpLight
Address of the IDirect3DRMLight interface that is the light source.

px, py, and *pz*
Plane that the shadow is to be projected on.

nx, ny, and *nz*
Normal to the plane that the shadow is to be projected on.

lplpShadow
Address of a pointer to be initialized with a valid pointer to the shadow visual, if the call succeeds.

IDirect3DRM::CreateTexture

```
HRESULT CreateTexture(LPD3DRMIMAGE lpImage,
    LPDIRECT3DRMTEXTURE* lplpD3DRMTexture);
```

Creates a texture from an image in memory.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpImage
Address of a D3DRMIMAGE structure describing the source for the texture.

lplpD3DRMTexture
Address that will be filled with a pointer to an IDirect3DRMTexture interface if the call succeeds.

The memory associated with the image is used each time the texture is rendered, rather than the memory being copied into Direct3DRM's buffers. This allows the image to be used both as a rendering target and as a texture.

IDirect3DRM::CreateTextureFromSurface

```
HRESULT CreateTextureFromSurface(LPDIRECTDRAWSURFACE lpDDS,
    LPDIRECT3DRMTEXTURE * lplpD3DRMTexture);
```

Creates a texture from a specified DirectDraw surface.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpDDS
Address of the DirectDrawSurface object containing the texture.

lplpD3DRMTexture
Address that will be filled with a pointer to an IDirect3DRMTexture interface if the call succeeds.

IDirect3DRM::CreateUserVisual

```
HRESULT CreateUserVisual(D3DRMUSERVISUALCALLBACK fn,
    LPVOID  lpArg, LPDIRECT3DRMUSERVISUAL * lplpD3DRMUV);
```

Creates an application-defined visual object, which can then be added to a scene and rendered by using an application-defined handler.

- Should return D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

fn

Application-defined D3DRMUSERVISUALCALLBACK callback function.

lpArg

Address of application-defined data passed to the callback function.

lplpD3DRMUV

Address that will be filled with a pointer to an IDirect3DRMUserVisual interface if the call succeeds.

IDirect3DRM::CreateViewport

```
HRESULT CreateViewport(LPDIRECT3DRMDEVICE lpDev,
    LPDIRECT3DRMFRAME lpCamera, DWORD dwXPos,
    DWORD dwYPos, DWORD dwWidth, DWORD dwHeight,
    LPDIRECT3DRMVIEWPORT* lplpD3DRMViewport);
```

Creates a viewport on a device with device coordinates (*dwXPos*, *dwYPos*) to (*dwXPos* + *dwWidth*, *dwYPos* + *dwHeight*).

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpDev
Device on which the viewport is to be created.

lpCamera
Frame that describes the position and direction of the view.

dwXPos, dwYPos, dwWidth, and *dwHeight*
Position and size of the viewport, in device coordinates.

lplpD3DRMViewport
Address that will be filled with a pointer to an IDirect3DRMViewport interface if the call succeeds.

The viewport displays objects in the scene that contains the camera, with the view direction and up vector taken from the camera.

IDirect3DRM::CreateWrap

```
HRESULT CreateWrap(D3DRMWRAPTYPE type, LPDIRECT3DRMFRAME lpRef,
    D3DVALUE ox, D3DVALUE oy, D3DVALUE oz, D3DVALUE dx, D3DVALUE dy,
    D3DVALUE dz, D3DVALUE ux, D3DVALUE uy, D3DVALUE uz, D3DVALUE ou,
    D3DVALUE ov, D3DVALUE su, D3DVALUE sv,
    LPDIRECT3DRMWRAP* lplpD3DRMWrap);
```

Creates a wrapping function that can be used to assign texture coordinates to faces and meshes. The vector [*ox oy oz*] gives the origin of the wrap, [*dx dy dz*] gives its z-axis, and [*ux uy uz*] gives its y-axis. The 2D vectors [*ou ov*] and [*su sv*] give an origin and scale factor in the texture applied to the result of the wrapping function.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

type
One of the members of the D3DRMWRAPTYPE enumerated type.

lpRef
Reference frame for the wrap.

ox, oy, and *oz*
Origin of the wrap.

dx, dy, and *dz*
The z-axis of the wrap.

ux, uy, and *uz*
The y-axis of the wrap.

ou and *ov*
Origin in the texture.

su and *sv*
Scale factor in the texture.

lplpD3DRMWrap
Address that will be filled with a pointer to an IDirect3DRMWrap interface if the call succeeds.

IDirect3DRM::EnumerateObjects

```
HRESULT EnumerateObjects(D3DRMOBJECTCALLBACK func, LPVOID lpArg);
```

Calls the callback function specified by the *func* parameter on each of the active Direct3DRM objects.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

func
Application-defined D3DRMOBJECTCALLBACK callback function to be called with each Direct3DRMObject object and the application-defined argument.

lpArg
Address of application-defined data passed to the callback function.

IDirect3DRM::GetDevices

```
HRESULT GetDevices(LPDIRECT3DRMDEVICEARRAY* lplpDevArray);
```

Returns all the Direct3DRM devices that have been created in the system.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpDevArray

Address of a pointer that will be filled with the resulting array of Direct3DRM devices. For information about the Direct3DRMDeviceArray object, see the IDirect3DRMDeviceArray interface.

IDirect3DRM::GetNamedObject

```
HRESULT GetNamedObject(const char * lpName,
    LPDIRECT3DRMOBJECT* lplpD3DRMObject);
```

Finds a Direct3DRMObject, given its name.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpName
Name of the object to be searched for.

lplpD3DRMObject
Address of a pointer to be initialized with a valid Direct3DRMObject pointer if the call succeeds.

IDirect3DRM::GetSearchPath

```
HRESULT GetSearchPath(DWORD * lpdwSize, LPSTR lpszPath);
```

Returns the current file search path.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpdwSize

Address of the number of returned path elements. This parameter cannot be NULL.

lpszPath

Address of a null-terminated string specifying the search path. If this parameter is NULL, the method returns the size of the required string in the location pointed to by the *lpdwSize* parameter.

IDirect3DRM::Load

```
HRESULT Load(LPVOID lpvObjSource, LPVOID lpvObjID,
    LPIID * lplpGUIDs, DWORD dwcGUIDs, D3DRMLOADOPTIONS d3drmLOFlags,
    D3DRMLOADCALLBACK d3drmLoadProc, LPVOID lpArgLP,
    D3DRMLOADTEXTURECALLBACK d3drmLoadTextureProc, LPVOID lpArgLTP,
    LPDIRECT3DRMFRAME lpParentFrame);
```

Loads an object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpvObjSource
Source for the object to be loaded. This source can be a file, resource, memory block, or stream, depending on the source flags specified in the *d3drmLOFlags* parameter.

lpvObjID
Object name or position to be loaded. The use of this parameter depends on the identifier flags specified in the *d3drmLOFlags* parameter. If the D3DRMLOAD_BYPOSITION flag is specified, this parameter is a pointer to a **DWORD** value that gives the object's order in the file. This parameter can be NULL.

lplpGUIDs
Address of an array of interface identifiers to be loaded. For example, if this parameter is a two-element array containing IID_IDirect3DMeshBuilder and IID_IDirect3DRMAnimationSet, this method loads all the animation sets and mesh-builder objects.

dwcGUIDs
Number of elements specified in the *lplpGUIDs* parameter.

d3drmLOFlags
Value of the D3DRMLOADOPTIONS type describing the load options.

d3drmLoadProc
A D3DRMLOADCALLBACK callback function called when the system reads the specified object.

lpArgLP
Address of application-defined data passed to the **D3DRMLOADCALLBACK** callback function.

d3drmLoadTextureProc
A D3DRMLOADTEXTURECALLBACK callback function called to load any textures used by an object.

lpArgLTP
Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

lpParentFrame
Address of a parent frame. This information is useful when loading Direct3DRMAnimationSet or Direct3DRMFrame objects because these objects would be created with a NULL parent otherwise. This value can be NULL.

IDirect3DRM::LoadTexture

```
HRESULT LoadTexture(const char * lpFileName,
    LPDIRECT3DRMTEXTURE* lplpD3DRMTexture);
```

Loads a texture from the specified file. This texture can have 8, 24, or 32 bits-per-pixel, and it should be in either the Windows bitmap (.bmp) or Portable Pixmap (.ppm) P6 format.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpFileName

Name of the required .bmp or .ppm file.

lplpD3DRMTexture

Address of a pointer to be initialized with a valid Direct3DRMTexture pointer if the call succeeds.

IDirect3DRM::LoadTextureFromResource

```
HRESULT LoadTextureFromResource(HRSRC rs,
    LPDIRECT3DRMTEXTURE * lplpD3DRMTexture);
```

Loads a texture from a specified resource.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rs
Handle of the resource.

lplpD3DRMTexture
Address of a pointer to be initialized with a valid Direct3DRMTexture object if the call succeeds.

IDirect3DRM::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRM object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRM.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRM::QueryInterface** method allows Direct3DRM objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRM::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRM object by 1. This method is part of the IUnknown interface inherited by Direct3DRM.

- Returns the new reference count of the object.

The Direct3DRM object deallocates itself when its reference count reaches 0. Use the IDirect3DRM::AddRef method to increase the reference count of the object by 1.

IDirect3DRM::SetDefaultTextureColors

```
HRESULT SetDefaultTextureColors(DWORD dwColors);
```

Sets the default colors to be used for a Direct3DRMTexture object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

dwColors

Number of colors.

This method affects the texture colors only when it is called before the IDirect3DRM::CreateTexture method; it has no effect on textures that have already been created.

IDirect3DRM::SetDefaultTextureShades

```
HRESULT SetDefaultTextureShades(DWORD dwShades);
```

Sets the default shades to be used for an Direct3DRMTexture object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

dwShades
Number of shades.

This method affects the texture shades only when it is called before the IDirect3DRM::CreateTexture method; it has no effect on textures that have already been created.

IDirect3DRM::SetSearchPath

```
HRESULT SetSearchPath(LPCSTR lpPath);
```

Sets the current file search path from a list of directories.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpPath

Address of a null-terminated string specifying the path to set as the current search path.

The default search path is taken from the value of the D3DPATH environment variable. If this is not set, the search path will be empty. When opening a file, the system first looks for the file in the current working directory and then checks every directory in the search path.

IDirect3DRM::Tick

```
HRESULT Tick(D3DVALUE d3dvalTick);
```

Performs the Direct3DRM system heartbeat. When this method is called, the positions of all moving frames are updated according to their current motion attributes, the scene is rendered to the current device, and relevant callback functions are called at their appropriate times. Control is returned when the rendering cycle is complete.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3dvalTick

Velocity and rotation step for the IDirect3DRMFrame::SetRotation and IDirect3DRMFrame::SetVelocity methods.

You can implement this method by using other Retained-Mode methods to allow more flexibility in rendering a scene.

IDirect3DRMAnimation Interface Method Groups

Applications use the methods of the **IDirect3DRMAnimation** interface to animate the position, orientation, and scaling of visuals, lights, and viewports. The methods can be organized into the following groups:

IUnknown	AddRef QueryInterface Release
Keys	AddPositionKey AddRotateKey AddScaleKey DeleteKey
Miscellaneous	SetFrame SetTime
Options	GetOptions SetOptions

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMAnimation object without affecting the functionality of the original interface. In addition, the **IDirect3DRMAnimation** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMAnimation object is obtained by calling the IDirect3DRM::CreateAnimation method.

IDirect3DRMAnimation::AddPositionKey

```
HRESULT AddPositionKey(D3DVALUE rvTime, D3DVALUE rvX,
    D3DVALUE rvY, D3DVALUE rvZ);
```

Adds a position key to the animation.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvTime

Time in the animation to store the position key. The time units are arbitrary and zero-based; a key whose *rvTime* value is 49 occurs exactly in the middle of an animation whose last key has an *rvTime* value of 99.

rvX, rvY, and *rvZ*

Position.

The transformation applied by this method is a translation. For information about the matrix mathematics involved in transformations, see 3D Transformations.

IDirect3DRMAnimation::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMAnimation object by 1. This method is part of the IUnknown interface inherited by Direct3DRMAnimation.

- Returns the new reference count of the object.

When the Direct3DRMAnimation object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMAnimation::Release method to decrease the reference count of the object by 1.

IDirect3DRMAnimation::AddRotateKey

```
HRESULT AddRotateKey(D3DVALUE rvTime, D3DRMQUATERNION *rqQuat);
```

Adds a rotate key to the animation.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvTime

Time in the animation to store the rotate key. The time units are arbitrary and zero-based; a key whose *rvTime* value is 49 occurs exactly in the middle of an animation whose last key has an *rvTime* value of 99.

rqQuat

Quaternion representing the rotation.

This method applies a rotation transformation. For information about the matrix mathematics involved in transformations, see 3D Transformations.

IDirect3DRMAnimation::AddScaleKey

```
HRESULT AddScaleKey(D3DVALUE rvTime, D3DVALUE rvX, D3DVALUE rvY,
    D3DVALUE rvZ);
```

Adds a scale key to the animation.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvTime

Time in the animation to store the scale key. The time units are arbitrary and zero-based; a key whose *rvTime* value is 49 occurs exactly in the middle of an animation whose last key has an *rvTime* value of 99.

rvX, rvY, and *rvZ*

Scale factor.

This method applies a scaling transformation. For information about the matrix mathematics involved in transformations, see 3D Transformations.

IDirect3DRMAnimation::DeleteKey

```
HRESULT DeleteKey(D3DVALUE rvTime);
```

Removes a key from an animation.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvTime
Time identifying the key that will be removed from the animation.

IDirect3DRMAnimation::GetOptions

```
D3DRMANIMATIONOPTIONS GetOptions();
```

Retrieves animation options.

- Returns the value of the D3DRMANIMATIONOPTIONS type describing the animation options.

IDirect3DRMAnimation::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMAnimation object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMAnimation.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **QueryInterface** method allows Direct3DRMAnimation objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMAnimation::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMAnimation object by 1. This method is part of the IUnknown interface inherited by Direct3DRMAnimation.

- Returns the new reference count of the object.

The Direct3DRMAnimation object deallocates itself when its reference count reaches 0. Use the IDirect3DRMAnimation::AddRef method to increase the reference count of the object by 1.

IDirect3DRMAnimation::SetFrame

```
HRESULT SetFrame(LPDIRECT3DRMFRAME lpD3DRMFrame);
```

Sets the frame for the animation.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMFrame
Address of a variable representing the frame to set for the animation.

IDirect3DRMAnimation::SetOptions

```
HRESULT SetOptions(D3DRMANIMATIONOPTIONS d3drmanimFlags);
```

Sets the animation options.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmanimFlags
Value of the D3DRMANIMATIONOPTIONS type describing the animation options.

IDirect3DRMAnimation::SetTime

```
HRESULT SetTime(D3DVALUE rvTime);
```

Sets the current time for this animation.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvTime
New current time for the animation. The time units are arbitrary and zero-based; a key whose *rvTime* value is 49 occurs exactly in the middle of an animation whose last key has an *rvTime* value of 99.

IDirect3DRMAnimationSet Interface Method Groups

Applications use the methods of the **IDirect3DRMAnimationSet** interface to group Direct3DRMAnimation objects together, which can simplify the playback of complex animation sequences. The methods can be organized into the following groups:

Adding, loading, and removing	AddAnimation DeleteAnimation Load
IUnknown	AddRef QueryInterface Release
Time	SetTime

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMAnimationSet object without affecting the functionality of the original interface. In addition, the **IDirect3DRMAnimationSet** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMAnimationSet object is obtained by calling the IDirect3DRM::CreateAnimationSet method.

IDirect3DRMAnimationSet::AddAnimation

```
HRESULT AddAnimation(LPDIRECT3DRMANIMATION lpD3DRMAnimation);
```

Adds an animation to the animation set.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMAnimation
Address of the Direct3DRMAnimation object to be added to the animation set.

IDirect3DRMAnimationSet::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMAnimationSet object by 1. This method is part of the IUnknown interface inherited by Direct3DRMAnimationSet.

- Returns the new reference count of the object.

When the Direct3DRMAnimationSet object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMAnimationSet::Release method to decrease the reference count of the object by 1.

IDirect3DRMAnimationSet::DeleteAnimation

```
HRESULT DeleteAnimation(LPDIRECT3DRMANIMATION lpD3DRMAnimation);
```

Removes a previously added animation from the animation set.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMAnimation
Address of the Direct3DRMAnimation object to be removed from the animation set.

IDirect3DRMAnimationSet::Load

```
HRESULT Load(LPVOID lpvObjSource, LPVOID lpvObjID,
    D3DRMLOADOPTIONS d3drmLOFlags,
    D3DRMLOADTEXTURECALLBACK d3drmLoadTextureProc, LPVOID lpArgLTP,
    LPDIRECT3DRMFRAME lpParentFrame);
```

Loads an animation set.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpvObjSource
Source for the object to be loaded. This source can be a file, resource, memory block, or stream, depending on the source flags specified in the *d3drmLOFlags* parameter.

lpvObjID
Object name or position to be loaded. The use of this parameter depends on the identifier flags specified in the *d3drmLOFlags* parameter. If the D3DRMLOAD_BYPOSITION flag is specified, this parameter is a pointer to a **DWORD** value that gives the object's order in the file. This parameter can be NULL.

d3drmLOFlags
Value of the D3DRMLOADOPTIONS type describing the load options.

d3drmLoadTextureProc
A D3DRMLOADTEXTURECALLBACK callback function called to load any textures used by the object.

lpArgLTP
Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

lpParentFrame
Address of a parent Direct3DRMFrame object. This prevents the frames referred to by the animation set from being created with a NULL parent.

By default, this method loads the first animation set in the file specified by the *lpvObjSource* parameter.

IDirect3DRMAnimationSet::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMAnimationSet object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMAnimationSet.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMAnimationSet::QueryInterface** method allows Direct3DRMAnimationSet objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMAnimationSet::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMAnimationSet object by 1. This method is part of the IUnknown interface inherited by Direct3DRMAnimationSet.

- Returns the new reference count of the object.

The Direct3DRMAnimationSet object deallocates itself when its reference count reaches 0. Use the IDirect3DRMAnimationSet::AddRef method to increase the reference count of the object by 1.

IDirect3DRMAnimationSet::SetTime

```
HRESULT SetTime(D3DVALUE rvTime);
```

Sets the time for this animation set.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvTime
New time.

IDirect3DRMDevice Interface Method Groups

Applications use the methods of the **IDirect3DRMDevice** interface to interact with the output device. The methods can be organized into the following groups:

Buffer counts	GetBufferCount SetBufferCount
Color models	GetColorModel
Dithering	GetDither SetDither
Initialization	Init InitFromClipper InitFromD3D
IUnknown	AddRef QueryInterface Release
Miscellaneous	GetDirect3DDevice GetHeight GetTrianglesDrawn GetViewports GetWidth GetWireframeOptions Update
Notifications	AddUpdateCallback DeleteUpdateCallback
Rendering quality	GetQuality SetQuality
Shading	GetShades SetShades
Texture quality	GetTextureQuality SetTextureQuality

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMDevice object without affecting the functionality of the original interface. In addition, the **IDirect3DRMDevice** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData

GetClassName
GetName
SetAppData
SetName

The Direct3DRMDevice object is obtained by calling the IDirect3DRM::CreateDevice method.

IDirect3DRMDevice::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMDevice object by 1. This method is part of the IUnknown interface inherited by Direct3DRMDevice.

- Returns the new reference count of the object.

When the Direct3DRMDevice object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMDevice::Release method to decrease the reference count of the object by 1.

IDirect3DRMDevice::AddUpdateCallback

```
HRESULT AddUpdateCallback(D3DRMUPDATECALLBACK d3drmUpdateProc, LPVOID arg);
```

Adds a callback function that alerts the application when a change occurs to the device. The system calls this callback function whenever the application calls the IDirect3DRMDevice::Update method.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmUpdateProc
Address of an application-defined callback function, D3DRMUPDATECALLBACK.

arg
Private data to be passed to the update callback function.

IDirect3DRMDevice::DeleteUpdateCallback

```
HRESULT DeleteUpdateCallback(D3DRMUPDATECALLBACK d3drmUpdateProc,
    LPVOID arg);
```

Removes an update callback function that was added by calling the IDirect3DRMDevice::AddUpdateCallback method.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmUpdateProc

Address of an application-defined callback function, D3DRMUPDATECALLBACK.

arg

Private data that was passed to the update callback function.

IDirect3DRMDevice::GetBufferCount

```
DWORD GetBufferCount();
```

Retrieves the value set in a call to the IDirect3DRMDevice::SetBufferCount method.

- Returns the number of buffers—one for single-buffering, two for double-buffering, and so on.

IDirect3DRMDevice::GetColorModel

```
D3DCOLORMODEL GetColorModel();
```

Retrieves the color model of a device.

- Returns a value from the D3DCOLORMODEL enumerated type that describes the Direct3D color model (RGB or monochrome).

IDirect3DRMDevice::GetDirect3DDevice

```
HRESULT GetDirect3DDevice(LPDIRECT3DDEVICE * lplpD3DDevice);
```

Retrieves a pointer to an Immediate-Mode device.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DDevice
Address of a pointer that is initialized with a pointer to an Immediate-Mode device object.

IDirect3DRMDevice::GetDither

```
BOOL GetDither();
```

Retrieves the dither flag for the device.

- Returns TRUE if the dither flag is set, or FALSE otherwise.

IDirect3DRMDevice::GetHeight

```
DWORD GetHeight();
```

Retrieves the height, in pixels, of a device. This method is a convenience function.

- Returns the height.

IDirect3DRMDevice::GetTrianglesDrawn

```
DWORD GetTrianglesDrawn();
```

Retrieves the number of triangles drawn to a device since its creation. This method is a convenience function.

- Returns the number of triangles.

The number of triangles includes those that were passed to the renderer but were not drawn because they were backfacing. The number does not include triangles that were rejected for lying outside of the viewing frustum.

IDirect3DRMDevice::GetQuality

```
D3DRMRENDERQUALITY GetQuality();
```

Retrieves the rendering quality for the device.

- Returns one or more of the members of the enumerated types represented by the D3DRMRENDERQUALITY type.

IDirect3DRMDevice::GetShades

```
DWORD GetShades();
```

Retrieves the number of shades in a ramp of colors used for shading.

- Returns the number of shades.

IDirect3DRMDevice::GetTextureQuality

```
D3DRMTEXTUREQUALITY GetTextureQuality();
```

Retrieves the current texture quality parameter for the device. Texture quality is relevant only for an RGB device.

- Returns one of the members of the D3DRMTEXTUREQUALITY enumerated type.

IDirect3DRMDevice::GetViewports

```
HRESULT GetViewports(LPDIRECT3DRMVIEWPORTARRAY* lplpViewports);
```

Constructs a Direct3DRMViewportArray object that represents the viewports currently constructed from the device.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpViewports
Address of a pointer that is initialized with a valid Direct3DRMViewportArray object if the call succeeds.

IDirect3DRMDevice::GetWidth

```
DWORD GetWidth();
```

Retrieves the width, in pixels, of a device. This method is a convenience function.

- Returns the width.

IDirect3DRMDevice::GetWireframeOptions

```
DWORD GetWireframeOptions();
```

Retrieves the wireframe options of a given device.

- Returns a bitwise **OR** of the following values:
  D3DRMWIREFRAME_CULL
  The backfacing faces are not drawn.
  D3DRMWIREFRAME_HIDDENLINE
  Wireframe-rendered lines are obscured by nearer objects.

IDirect3DRMDevice::Init

```
HRESULT Init(ULONG width, ULONG height);
```

Not implemented on the Windows platform.

IDirect3DRMDevice::InitFromClipper

```
HRESULT InitFromClipper(LPDIRECTDRAWCLIPPER lpDDClipper,
    LPGUID lpGUID, int width, int height);
```

Initializes a device from a specified DirectDrawClipper object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpDDClipper
Address of the DirectDrawClipper object to use as an initializer.

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

width and *height*
Width and height of the device.

IDirect3DRMDevice::InitFromD3D

```
HRESULT InitFromD3D(LPDIRECT3D lpD3D, LPDIRECT3DDEVICE lpD3DIMDev);
```

Initializes a Retained-Mode device from a specified Direct3D Immediate-Mode object and Immediate-Mode device.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3D
Address of the Direct3D Immediate-Mode object to use to initialize the Retained-Mode device.

lpD3DIMDev
Address of the Immediate-Mode device to use to initialize the Retained-Mode device.

IDirect3DRMDevice::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMDevice object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMDevice.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMDevice::QueryInterface** method allows Direct3DRMDevice objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMDevice::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMDevice object by 1. This method is part of the IUnknown interface inherited by Direct3DRMDevice.

- Returns the new reference count of the object.

The Direct3DRMDevice object deallocates itself when its reference count reaches 0. Use the IDirect3DRMDevice::AddRef method to increase the reference count of the object by 1.

IDirect3DRMDevice::SetBufferCount

```
HRESULT SetBufferCount(DWORD dwCount);
```

Sets the number of buffers currently being used by the application.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

dwCount

Specifies the number of buffers—one for single-buffering, two for double-buffering, and so on. The default value is 1, which is correct only for single-buffered window operation.

An application that employs double-buffering or triple-buffering must use this method to inform the system of how many buffers it is using so that the system can calculate how much of the window to clear and update on each frame.

IDirect3DRMDevice::SetDither

```
HRESULT SetDither(BOOL bDither);
```

Sets the dither flag for the device.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

bDither
New dithering mode for the device. The default is TRUE.

IDirect3DRMDevice::SetQuality

```
HRESULT SetQuality (D3DRMRENDERQUALITY rqQuality);
```

Sets the rendering quality of a device

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rqQuality

One or more of the members of the enumerated types represented by the D3DRMRENDERQUALITY type. The default setting is D3DRMRENDER_FLAT.

The rendering quality is the maximum quality at which rendering can take place on the rendering surface of that device. Each mesh can have its own quality, but the maximum quality available for a mesh is that of the device. Different devices can have different qualities. For example, previewing devices usually have a lower quality, while devices used for final viewing usually have a higher quality.

IDirect3DRMDevice::SetShades

```
HRESULT SetShades(DWORD ulShades);
```

Sets the number of shades in a ramp of colors used for shading.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

ulShades
New number of shades. This parameter must be a power of 2. The default is 32.

IDirect3DRMDevice::SetTextureQuality

```
HRESULT SetTextureQuality(D3DRMTEXTUREQUALITY tqTextureQuality);
```

Sets the texture quality for the device.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

tqTextureQuality
One of the members of the D3DRMTEXTUREQUALITY enumerated type. The default is D3DRMTEXTURE_NEAREST.

IDirect3DRMDevice::Update

```
HRESULT Update();
```

Copies the image that has been rendered to the display. It also provides a heartbeat function to the device driver.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Each call to this method causes the system to call an application-defined callback function, D3DRMUPDATECALLBACK. To add a callback function, use the IDirect3DRMDevice::AddUpdateCallback method.

IDirect3DRMFace Interface Method Groups

Applications use the methods of the **IDirect3DRMFace** interface to interact with a single polygon in a mesh. The methods can be organized into the following groups:

Color	GetColor SetColor SetColorRGB
IUnknown	AddRef QueryInterface Release
Materials	GetMaterial SetMaterial
Textures	GetTexture GetTextureCoordinateIndex GetTextureCoordinates GetTextureTopology SetTexture SetTextureCoordinates SetTextureTopology
Vertices and normals	AddVertex AddVertexAndNormalIndexed GetNormal GetVertex GetVertexCount GetVertexIndex GetVertices

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMFace object without affecting the functionality of the original interface. In addition, the **IDirect3DRMFace** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMFace object is obtained by calling the IDirect3DRM::CreateFace method.

IDirect3DRMFace::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMFace object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFace.

- Returns the new reference count of the object.

When the Direct3DRMFace object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMFace::Release method to decrease the reference count of the object by 1.

IDirect3DRMFace::AddVertex

```
HRESULT AddVertex(D3DVALUE x, D3DVALUE y, D3DVALUE z);
```

Adds a vertex to a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

x, y, and *z*
The x, y, and z components of the position of the new vertex.

IDirect3DRMFace::AddVertexAndNormalIndexed

```
HRESULT AddVertexAndNormalIndexed(DWORD vertex, DWORD normal);
```

Adds a vertex and a normal to a Direct3DRMFace object, using an index for the vertex and an index for the normal in the containing mesh builder. The face, vertex, and normal must already be part of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

vertex and *normal*
Indexes of the vertex and normal to add.

IDirect3DRMFace::GetColor

```
D3DCOLOR GetColor();
```

Retrieves the color of a Direct3DRMFace object.

- Returns the color.

IDirect3DRMFace::GetMaterial

```
HRESULT GetMaterial(LPDIRECT3DRMMATERIAL* lplpMaterial);
```

Retrieves the material of a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpMaterial
Address of a variable that will be filled with a pointer to the Direct3DRMMaterial object applied to the face.

IDirect3DRMFace::GetNormal

```
HRESULT GetNormal(D3DVECTOR *lpNormal);
```

Retrieves the normal of a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpNormal
Address of a D3DVECTOR structure that will be filled with the normal vector of the face.

IDirect3DRMFace::GetTexture

```
HRESULT GetTexture(LPDIRECT3DRMTEXTURE* lplpTexture);
```

Retrieves the Direct3DRMTexture object applied to a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpTexture
Address of a variable that will be filled with a pointer to the texture applied to the face.

IDirect3DRMFace::GetTextureCoordinateIndex

```
int GetTextureCoordinateIndex(DWORD dwIndex);
```

Retrieves the index of the vertex for texture coordinates in the face's mesh. This index corresponds to the index specified in the *dwIndex* parameter.

- Returns the index.

dwIndex
Index within the face of the vertex.

IDirect3DRMFace::GetTextureCoordinates

```
HRESULT GetTextureCoordinates(DWORD index, D3DVALUE *lpU,
    D3DVALUE *lpV);
```

Retrieves the texture coordinates of a vertex in a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the vertex.

lpU and *lpV*
Addresses of variables that are filled with the texture coordinates of the vertex.

IDirect3DRMFace::GetTextureTopology

```
HRESULT GetTextureTopology(BOOL *lpU, BOOL *lpV);
```

Retrieves the texture topology of a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpU and *lpV*
Addresses of variables that are set or cleared depending on how the cylindrical wrapping flags are set for the face.

IDirect3DRMFace::GetVertex

```
HRESULT GetVertex(DWORD index, D3DVECTOR *lpPosition,
    D3DVECTOR *lpNormal);
```

Retrieves the position and normal of a vertex in a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the vertex.

lpPosition and *lpNormal*
Addresses of D3DVECTOR structures that will be filled with the position and normal of the vertex, respectively.

IDirect3DRMFace::GetVertexCount

```
int GetVertexCount();
```

Retrieves the number of vertices in a Direct3DRMFace object.

- Returns the number of vertices.

IDirect3DRMFace::GetVertexIndex

```
int GetVertexIndex (DWORD dwIndex);
```

Retrieves the index of the vertex in the face's mesh. This index corresponds to the index specified in the *dwIndex* parameter.

- Returns the index.

dwIndex
Index within the face of the vertex.

IDirect3DRMFace::GetVertices

```
HRESULT GetVertices(DWORD *lpdwVertexCount, D3DVECTOR *lpPosition,
    D3DVECTOR *lpNormal);
```

Retrieves the position and normal of each vertex in a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpdwVertexCount
Address of a variable that is filled with the number of vertices. This parameter cannot be NULL.

lpPosition and *lpNormal*
Arrays of D3DVECTOR structures that will be filled with the positions and normal vectors of the vertices, respectively. If both of these parameters are NULL, the method will fill the *lpdwVertexCount* parameter with the number of vertices that will be retrieved.

IDirect3DRMFace::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMFace object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMFace.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that is filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMFace::QueryInterface** method allows Direct3DRMFace objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMFace::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMFace object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFace.

- Returns the new reference count of the object.

The Direct3DRMFace object deallocates itself when its reference count reaches 0. Use the IDirect3DRMFace::AddRef method to increase the reference count of the object by 1.

IDirect3DRMFace::SetColor

```
HRESULT SetColor(D3DCOLOR color);
```

Sets a Direct3DRMFace object to a given color.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

color
Color to set.

IDirect3DRMFace::SetColorRGB

```
HRESULT SetColorRGB(D3DVALUE red, D3DVALUE green, D3DVALUE blue);
```

Sets a Direct3DRMFace object to a given color.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

red, green, and *blue*
The red, green, and blue components of the color.

IDirect3DRMFace::SetMaterial

```
HRESULT SetMaterial(LPDIRECT3DRMMATERIAL lpD3DRMMaterial);
```

Sets the material of a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMMaterial
Address of the material.

IDirect3DRMFace::SetTexture

```
HRESULT SetTexture(LPDIRECT3DRMTEXTURE lpD3DRMTexture);
```

Sets the texture of a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMTexture
Address of the texture.

IDirect3DRMFace::SetTextureCoordinates

```
HRESULT SetTextureCoordinates(DWORD vertex, D3DVALUE u, D3DVALUE v);
```

Sets the texture coordinates of a specified vertex in a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

vertex
Index of the vertex to be set. For example, if the face were a triangle, the possible vertex indices would be 0, 1, and 2.

u and *v*
Texture coordinates to assign to the specified vertex.

IDirect3DRMFace::SetTextureTopology

```
HRESULT SetTextureTopology(BOOL cylU, BOOL cylV);
```

Sets the texture topology of a Direct3DRMFace object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

cylU and *cylV*
Specify whether the texture has a cylindrical topology in the u and v dimensions.

IDirect3DRMFrame Interface Method Groups

Applications use the methods of the **IDirect3DRMFrame** interface to interact with frames—an object's frame of reference. The methods can be organized into the following groups:

Background	GetSceneBackground GetSceneBackgroundDepth SetSceneBackground SetSceneBackgroundDepth SetSceneBackgroundImage SetSceneBackgroundRGB
Color	GetColor SetColor SetColorRGB
Fog	GetSceneFogColor GetSceneFogEnable GetSceneFogMode GetSceneFogParams SetSceneFogColor SetSceneFogEnable SetSceneFogMode SetSceneFogParams
Hierarchies	AddChild DeleteChild GetChildren GetParent GetScene
IUnknown	AddRef QueryInterface Release
Lighting	AddLight DeleteLight GetLights
Loading	Load
Material modes	GetMaterialMode SetMaterialMode
Positioning and movement	AddMoveCallback AddRotation AddScale AddTranslation DeleteMoveCallback

	GetOrientation GetPosition GetRotation GetVelocity LookAt Move SetOrientation SetPosition SetRotation SetVelocity
Sorting	GetSortMode GetZbufferMode SetSortMode SetZbufferMode
Textures	GetTexture GetTextureTopology SetTexture SetTextureTopology
Transformations	AddTransform GetTransform InverseTransform Transform
Visual objects	AddVisual DeleteVisual GetVisuals

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMFrame object without affecting the functionality of the original interface. In addition, the **IDirect3DRMFrame** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMFrame object is obtained by calling the IDirect3DRM::CreateFrame method.

IDirect3DRMFrame::AddChild

```
HRESULT AddChild(LPDIRECT3DRMFRAME lpD3DRMFrameChild);
```

Adds a child frame to a frame hierarchy.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMFrameChild
Address of the Direct3DRMFrame object that will be added as a child.

If the frame being added as a child already has a parent, this method removes it from its previous parent before adding it to the new parent.

IDirect3DRMFrame::AddLight

```
HRESULT AddLight(LPDIRECT3DRMLIGHT lpD3DRMLight);
```

Adds a light to a frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMLight
Address of a variable that represents the Direct3DRMLight object to be added to the frame.

IDirect3DRMFrame::AddMoveCallback

```
HRESULT AddMoveCallback(D3DRMFRAMEMOVECALLBACK d3drmFMC, VOID * lpArg);
```

Adds a callback function for special movement processing.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmFMC
Application-defined D3DRMFRAMEMOVECALLBACK callback function.

lpArg
Application-defined data to be passed to the callback function.

IDirect3DRMFrame::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMFrame object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFrame.

- Returns the new reference count of the object.

When the Direct3DRMFrame object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMFrame::Release method to decrease the reference count of the object by 1.

IDirect3DRMFrame::AddRotation

```
HRESULT AddRotation(D3DRMCOMBINETYPE rctCombine, D3DVALUE rvX,
    D3DVALUE rvY, D3DVALUE rvZ, D3DVALUE rvTheta);
```

Adds a rotation about (*rvX*, *rvY*, *rvZ*) by the number of radians specified in *rvTheta*.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rctCombine

A member of the D3DRMCOMBINETYPE enumerated type that specifies how to combine the new rotation with any current frame transformation.

rvX, rvY, and *rvZ*

Axis about which to rotate.

rvTheta

Angle of rotation, in radians.

The specified rotation changes the matrix only for the frame identified by this IDirect3DRMFrame interface. This method changes the objects in the frame only once, unlike IDirect3DRMFrame::SetRotation, which changes the matrix with every render tick.

IDirect3DRMFrame::AddScale

```
HRESULT AddScale(D3DRMCOMBINETYPE rctCombine, D3DVALUE rvX,
    D3DVALUE rvY, D3DVALUE rvZ);
```

Scales a frame's local transformation by (*rvX*, *rvY*, *rvZ*).

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rctCombine

Member of the D3DRMCOMBINETYPE enumerated type that specifies how to combine the new scale with any current frame transformation.

rvX, rvY, and *rvZ*

Define the scale factors in the x, y, and z directions.

The specified transformation changes the matrix only for the frame identified by this IDirect3DRMFrame interface.

IDirect3DRMFrame::AddTransform

```
HRESULT AddTransform(D3DRMCOMBINETYPE rctCombine,
    D3DRMMATRIX4D rmMatrix);
```

Transforms the local coordinates of the frame by the given affine transformation according to the value of the *rctCombine* parameter.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rctCombine

Member of the D3DRMCOMBINETYPE enumerated type that specifies how to combine the new transformation with any current transformation.

rmMatrix

Member of the D3DRMMATRIX4D array that defines the transformation matrix to be combined.

Although a 4-by-4 matrix is given, the last column must be the transpose of [0 0 0 1] for the transformation to be affine.

The specified transformation changes the matrix only for the frame identified by this IDirect3DRMFrame interface.

IDirect3DRMFrame::AddTranslation

```
HRESULT AddTranslation(D3DRMCOMBINETYPE rctCombine, D3DVALUE rvX,
    D3DVALUE rvY, D3DVALUE rvZ);
```

Adds a translation by (*rvX*, *rvY*, *rvZ*) to a frame's local coordinate system.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rctCombine

Member of the D3DRMCOMBINETYPE enumerated type that specifies how to combine the new translation with any current translation.

rvX, rvY, and *rvZ*

Define the position changes in the x, y, and z directions.

The specified translation changes the matrix only for the frame identified by this IDirect3DRMFrame interface.

IDirect3DRMFrame::AddVisual

```
HRESULT AddVisual(LPDIRECT3DRMVISUAL lpD3DRMVisual);
```

Adds a visual object to a frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMVisual

Address of a variable that represents the Direct3DRMVisual object to be added to the frame.

Visual objects include meshes and textures. When a visual object is added to a frame, it becomes visible if the frame is in view. The visual object is referenced by the frame.

IDirect3DRMFrame::DeleteChild

```
HRESULT DeleteChild(LPDIRECT3DRMFRAME lpChild);
```

Removes a frame from the hierarchy. If the frame is not referenced, it is destroyed along with any child frames, lights, and meshes.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpChild
Address of a variable that represents the Direct3DRMFrame object to be used as the child.

IDirect3DRMFrame::DeleteLight

```
HRESULT DeleteLight(LPDIRECT3DRMLIGHT lpD3DRMLight);
```

Removes a light from a frame, destroying it if it is no longer referenced. When a light is removed from a frame, it no longer affects meshes in the scene its frame was in.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMLight
Address of a variable that represents the Direct3DRMLight object to be removed.

IDirect3DRMFrame::DeleteMoveCallback

```
HRESULT DeleteMoveCallback(D3DRMFRAMEMOVECALLBACK d3drmFMC,
    VOID * lpArg);
```

Removes a callback function that performed special movement processing.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmFMC
Application-defined D3DRMFRAMEMOVECALLBACK callback function.

lpArg
Application-defined data that was passed to the callback function.

IDirect3DRMFrame::DeleteVisual

```
HRESULT DeleteVisual(LPDIRECT3DRMVISUAL lpD3DRMVisual);
```

Removes a visual object from a frame, destroying it if it is no longer referenced.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMVisual
Address of a variable that represents the Direct3DRMVisual object to be removed.

IDirect3DRMFrame::GetChildren

```
HRESULT GetChildren(LPDIRECT3DRMFRAMEARRAY* lplpChildren);
```

Retrieves a list of child frames in the form of a Direct3DRMFrameArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpChildren
Address of a pointer to be initialized with a valid Direct3DRMFrameArray pointer if the call succeeds.

IDirect3DRMFrame::GetColor

```
D3DCOLOR GetColor();
```

Retrieves the color of the frame.

- Returns the color of the Direct3DRMFrame object.

IDirect3DRMFrame::GetLights

```
HRESULT GetLights(LPDIRECT3DRMLIGHTARRAY* lplpLights);
```

Retrieves a list of lights in the frame in the form of a Direct3DRMLightArray object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpLights
Address of a pointer to be initialized with a valid Direct3DRMLightArray pointer if the call succeeds.

IDirect3DRMFrame::GetMaterialMode

```
D3DRMMATERIALMODE GetMaterialMode();
```

Retrieves the material mode of the frame.

- Returns a member of the D3DRMMATERIALMODE enumerated type that specifies the current material mode.

IDirect3DRMFrame::GetOrientation

```
HRESULT GetOrientation(LPDIRECT3DRMFRAME lpRef, LPD3DVECTOR lprvDir,
    LPD3DVECTOR lprvUp);
```

Retrieves the orientation of a frame relative to the given reference frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef
Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

lprvDir and *lprvUp*
Addresses of D3DVECTOR structures that will be filled with the directions of the frame's z- and y-axes respectively.

IDirect3DRMFrame::GetParent

```
HRESULT GetParent(LPDIRECT3DRMFRAME* lplpParent);
```

Retrieves the parent frame of the current frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpParent

Address of a pointer that will be filled with the pointer to the Direct3DRMFrame object representing the frame's parent. If the current frame is the root, this pointer will be NULL when the method returns.

IDirect3DRMFrame::GetPosition

```
HRESULT GetPosition(LPDIRECT3DRMFRAME lpRef, LPD3DVECTOR lprvPos);
```

Retrieves the position of a frame relative to the given reference frame (for example, this method retrieves the distance of the frame from the reference). The distance is stored in the *lprvPos* parameter as a vector rather than as a linear measure.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef

Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

lprvPos

Address of a D3DVECTOR structure that will be filled with the frame's position.

IDirect3DRMFrame::GetRotation

```
HRESULT GetRotation(LPDIRECT3DRMFRAME lpRef, LPD3DVECTOR lprvAxis,
    LPD3DVALUE lprvTheta);
```

Retrieves the rotation of the frame relative to the given reference frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef
Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

lprvAxis
Address of a D3DVECTOR structure that will be filled with the frame's axis of rotation.

lprvTheta
Address of a variable that will be the frame's rotation, in radians.

IDirect3DRMFrame::GetScene

```
HRESULT GetScene(LPDIRECT3DRMFRAME* lplpRoot);
```

Retrieves the root frame of the hierarchy containing the given frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpRoot

Address of the pointer that will be filled with the pointer to the Direct3DRMFrame object representing the scene's root frame.

IDirect3DRMFrame::GetSceneBackground

```
D3DCOLOR GetSceneBackground();
```

Retrieves the background color of a scene.

- Returns the color.

IDirect3DRMFrame::GetSceneBackgroundDepth

```
HRESULT GetSceneBackgroundDepth(
    LPDIRECTDRAWSURFACE * lplpDDSurface);
```

Retrieves the current background-depth buffer for the scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpDDSurface

Address of a pointer that will be initialized with the address of a DirectDraw surface representing the current background-depth buffer.

IDirect3DRMFrame::GetSceneFogColor

```
D3DCOLOR GetSceneFogColor();
```

Retrieves the fog color of a scene.

- Returns the fog color.

IDirect3DRMFrame::GetSceneFogEnable

```
BOOL GetSceneFogEnable();
```

Returns whether fog is currently enabled for this scene.

- Returns TRUE if fog is enabled, and FALSE otherwise.

IDirect3DRMFrame::GetSceneFogMode

```
D3DRMFOGMODE GetSceneFogMode();
```

Returns the current fog mode for this scene.

- Returns a member of the D3DRMFOGMODE enumerated type that specifies the current fog mode.

IDirect3DRMFrame::GetSceneFogParams

```
HRESULT GetSceneFogParams(D3DVALUE * lprvStart, D3DVALUE * lprvEnd,
    D3DVALUE * lprvDensity);
```

Retrieves the current fog parameters for this scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lprvStart, lprvEnd, and *lprvDensity*
Addresses of variables that will be the fog start, end, and density values.

IDirect3DRMFrame::GetSortMode

```
D3DRMSORTMODE GetSortMode();
```

Retrieves the sorting mode used to process child frames.

- Returns the member of the D3DRMSORTMODE enumerated type that specifies the sorting mode.

IDirect3DRMFrame::GetTexture

```
HRESULT GetTexture(LPDIRECT3DRMTEXTURE* lplpTexture);
```

Retrieves the texture of the given frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpTexture
Address of the pointer that will be filled with the address of the Direct3DRMTexture object representing the frame's texture.

IDirect3DRMFrame::GetTextureTopology

```
HRESULT GetTextureTopology(BOOL * lpbWrap_u, BOOL * lpbWrap_v);
```

Retrieves the topological properties of a texture when mapped onto objects in the given frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpbWrap_u and *lpbWrap_v*
Addresses of variables that are set to TRUE if the texture is mapped in the u and v directions, respectively.

IDirect3DRMFrame::GetTransform

```
HRESULT GetTransform(D3DRMMATRIX4D rmMatrix);
```

Retrieves the local transformation of the frame as a 4-by-4 affine matrix.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rmMatrix

A D3DRMMATRIX4D array that will be filled with the frame's transformation. Because this is an array, this value is actually an address.

IDirect3DRMFrame::GetVelocity

```
HRESULT GetVelocity(LPDIRECT3DRMFRAME lpRef, LPD3DVECTOR lprvVel,
    BOOL fRotVel);
```

Retrieves the velocity of the frame relative to the given reference frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef

Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

lprvVel

Address of a D3DVECTOR structure that will be filled with the frame's velocity.

fRotVel

Flag specifying whether the rotational velocity of the object is taken into account when retrieving the linear velocity. If this parameter is TRUE, the object's rotational velocity is included in the calculation.

IDirect3DRMFrame::GetVisuals

```
HRESULT GetVisuals(LPDIRECT3DRMVISUALARRAY* lplpVisuals);
```

Retrieves a list of visuals in the frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpVisuals
Address of a pointer to be initialized with a valid Direct3DRMVisualArray pointer if the call succeeds.

IDirect3DRMFrame::GetZbufferMode

```
D3DRMZBUFFERMODE GetZbufferMode();
```

Retrieves the z-buffer mode; that is, whether z-buffering is enabled or disabled.

- Returns one of the members of the D3DRMZBUFFERMODE enumerated type.

IDirect3DRMFrame::InverseTransform

```
HRESULT InverseTransform(D3DVECTOR *lprvDst, D3DVECTOR *lprvSrc);
```

Transforms the vector in the *lprvSrc* parameter in world coordinates to frame coordinates, and returns the result in the *lprvDst* parameter.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lprvDst
Address of a D3DVECTOR structure that will be filled with the result of the transformation.

lprvSrc
Address of a **D3DVECTOR** structure that is the source of the transformation.

IDirect3DRMFrame::Load

```
HRESULT Load(LPVOID lpvObjSource, LPVOID lpvObjID,
    D3DRMLOADOPTIONS d3drmLOFlags,
    D3DRMLOADTEXTURECALLBACK d3drmLoadTextureProc, LPVOID lpArgLTP);
```

Loads a Direct3DRMFrame object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpvObjSource

Source for the object to be loaded. This source can be a file, resource, memory block, or stream, depending on the source flags specified in the *d3drmLOFlags* parameter.

lpvObjID

Object name or position to be loaded. The use of this parameter depends on the identifier flags specified in the *d3drmLOFlags* parameter. If the D3DRMLOAD_BYPOSITION flag is specified, this parameter is a pointer to a **DWORD** value that gives the object's order in the file. This parameter can be NULL.

d3drmLOFlags

Value of the D3DRMLOADOPTIONS type describing the load options.

d3drmLoadTextureProc

A D3DRMLOADTEXTURECALLBACK callback function called to load any textures used by the object.

lpArgLTP

Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

By default, this method loads the first frame hierarchy in the file specified by the *lpvObjSource* parameter. The frame that calls this method is used as the parent of the new frame hierarchy.

IDirect3DRMFrame::LookAt

```
HRESULT LookAt(LPDIRECT3DRMFRAME lpTarget, LPDIRECT3DRMFRAME lpRef,
    D3DRMFRAMECONSTRAINT rfcConstraint);
```

Faces the frame toward the target frame, relative to the given reference frame, locking the rotation by the given constraints.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpTarget and *lpRef*

Addresses of variables that represent the Direct3DRMFrame objects to be used as the target and reference, respectively.

rfcConstraint

Member of the D3DRMFRAMECONSTRAINT enumerated type that specifies the axis of rotation to constrain.

IDirect3DRMFrame::Move

```
HRESULT Move(D3DVALUE delta);
```

Applies the rotations and velocities for all frames in the given hierarchy.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

delta
Amount to change the velocity and rotation.

IDirect3DRMFrame::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMFrame object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMFrame.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMFrame::QueryInterface** method allows Direct3DRMFrame objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMFrame::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMFrame object by 1. This method is part of the IUnknown interface inherited by Direct3DRMFrame.

- Returns the new reference count of the object.

The Direct3DRMFrame object deallocates itself when its reference count reaches 0. Use the IDirect3DRMFrame::AddRef method to increase the reference count of the object by 1.

IDirect3DRMFrame::SetColor

```
HRESULT SetColor(D3DCOLOR rcColor);
```

Sets the color of the frame. This color is used for meshes in the frame when the D3DRMMATERIALMODE enumerated type is D3DRMMATERIAL_FROMFRAME.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rcColor
New color for the frame.

IDirect3DRMFrame::SetColorRGB

```
HRESULT SetColorRGB(D3DVALUE rvRed, D3DVALUE rvGreen,
    D3DVALUE rvBlue);
```

Sets the color of the frame. This color is used for meshes in the frame when the D3DRMMATERIALMODE enumerated type is D3DRMMATERIAL_FROMFRAME.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvRed, rvGreen, and *rvBlue*
New color for the frame. Each component of the color should be in the range 0 to 1.

IDirect3DRMFrame::SetMaterialMode

```
HRESULT SetMaterialMode(D3DRMMATERIALMODE rmmMode);
```

Sets the material mode for a frame. The material mode determines the source of material information for visuals rendered with the frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rmmMode
One of the members of the D3DRMMATERIALMODE enumerated type.

IDirect3DRMFrame::SetOrientation

```
HRESULT SetOrientation(LPDIRECT3DRMFRAME lpRef, D3DVALUE rvDx,
    D3DVALUE rvDy, D3DVALUE rvDz, D3DVALUE rvUx, D3DVALUE rvUy,
    D3DVALUE rvUz);
```

Aligns a frame so that its z-direction points along the direction vector [*rvDx, rvDy, rvDz*] and its y-direction aligns with the vector [*rvUx, rvUy, rvUz*].

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef
Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

rvDx, rvDy, and *rvDz*
New z-axis for the frame.

rvUx, rvUy, and *rvUz*
New y-axis for the frame.

The default orientation of a frame has a direction vector of [0, 0, 1] and an up vector of [0, 1, 0].

If [*rvUx, rvUy, rvUz*] is parallel to [*rvDx, rvDy, rvDz*], the D3DRMERR_BADVALUE error value is returned; otherwise, the [*rvUx, rvUy, rvUz*] vector passed is projected onto the plane that is perpendicular to [*rvDx, rvDy, rvDz*].

IDirect3DRMFrame::SetPosition

```
HRESULT SetPosition(LPDIRECT3DRMFRAME lpRef, D3DVALUE rvX, D3DVALUE rvY,
    D3DVALUE rvZ);
```

Sets the position of a frame relative to the frame of reference. It places the frame a distance of [*rvX, rvY, rvZ*] from the reference. When a child frame is created within a parent, it is placed at [0, 0, 0] in the parent frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef

Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

rvX, rvY, and *rvZ*

New position for the frame.

IDirect3DRMFrame::SetRotation

```
HRESULT SetRotation(LPDIRECT3DRMFRAME lpRef, D3DVALUE rvX, D3DVALUE rvY,
    D3DVALUE rvZ, D3DVALUE rvTheta);
```

Sets a frame rotating by the given angle around the given vector at each call to the IDirect3DRM::Tick or IDirect3DRMFrame::Move method. The direction vector [*rvX, rvY, rvZ*] is defined in the reference frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef

Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

rvX, rvY, and *rvZ*

Vector about which rotation occurs.

rvTheta

Rotation angle, in radians.

The specified rotation changes the matrix with every render tick, unlike the IDirect3DRMFrame::AddRotation method, which changes the objects in the frame only once.

IDirect3DRMFrame::SetSceneBackground

```
HRESULT SetSceneBackground(D3DCOLOR rcColor);
```

Sets the background color of a scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rcColor
New color for the background.

IDirect3DRMFrame::SetSceneBackgroundDepth

```
HRESULT SetSceneBackgroundDepth(LPDIRECTDRAWSURFACE lpImage);
```

Specifies a background-depth buffer for a scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpImage

Address of a DirectDraw surface that will store the new background depth for the scene.

The image must have a depth of 16. If the image and viewport sizes are different, the image is scaled first. For best performance when animating the background-depth buffer, the image should be the same size as the viewport. This enables the depth buffer to be updated directly from the image memory without incurring extra overhead.

IDirect3DRMFrame::SetSceneBackgroundImage

```
HRESULT SetSceneBackgroundImage(LPDIRECT3DRMTEXTURE lpTexture);
```

Specifies a background image for a scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpTexture
Address of a Direct3DRMTexture object that will contain the new background scene.

If the image is a different size or color depth than the viewport, the image will first be scaled or converted to the correct depth. For best performance when animating the background, the image should be the same size and color depth. This enables the background to be rendered directly from the image memory without incurring any extra overhead.

IDirect3DRMFrame::SetSceneBackgroundRGB

```
HRESULT SetSceneBackgroundRGB(D3DVALUE rvRed, D3DVALUE rvGreen,
    D3DVALUE rvBlue);
```

Sets the background color of a scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvRed, rvGreen, and *rvBlue*
New color for the background.

IDirect3DRMFrame::SetSceneFogColor

```
HRESULT SetSceneFogColor(D3DCOLOR rcColor);
```

Sets the fog color of a scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rcColor
New color for the fog.

IDirect3DRMFrame::SetSceneFogEnable

```
HRESULT SetSceneFogEnable(BOOL bEnable);
```

Sets the fog enable state.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

bEnable
New fog enable state.

IDirect3DRMFrame::SetSceneFogMode

```
HRESULT SetSceneFogMode(D3DRMFOGMODE rfMode);
```

Sets the fog mode.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rfMode
One of the members of the D3DRMFOGMODE enumerated type, specifying the new fog mode.

IDirect3DRMFrame::SetSceneFogParams

```
HRESULT SetSceneFogParams(D3DVALUE rvStart, D3DVALUE rvEnd,
    D3DVALUE rvDensity);
```

Sets the current fog parameters for this scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvStart and *rvEnd*

Fog start and end points for linear fog mode. These settings determine the distance from the camera at which fog effects first become visible and the distance at which fog reaches its maximum density.

rvDensity

Fog density for the exponential fog modes. This value should be in the range 0 through 1.

IDirect3DRMFrame::SetSortMode

```
HRESULT SetSortMode(D3DRMSORTMODE d3drmSM);
```

Sets the sorting mode used to process child frames. You can use this method to change the properties of hidden-surface-removal algorithms.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmSM

One of the members of the D3DRMSORTMODE enumerated type, specifying the sorting mode. The default value is D3DRMSORT_FROMPARENT.

IDirect3DRMFrame::SetTexture

```
HRESULT SetTexture(LPDIRECT3DRMTEXTURE lpD3DRMTexture);
```

Sets the texture of the frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMTexture
Address of a variable that represents the Direct3DRMTexture object to be used.

The texture is used for meshes in the frame when the D3DRMMATERIALMODE enumerated type is D3DRMMATERIAL_FROMFRAME. To disable the frame's texture, use a NULL texture.

IDirect3DRMFrame::SetTextureTopology

```
HRESULT SetTextureTopology(BOOL bWrap_u, BOOL bWrap_v);
```

Defines the topological properties of the texture coordinates across objects in the frame.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

bWrap_u and *bWrap_v*
Variables that are set to TRUE to map the texture in the u- and v-directions, respectively.

IDirect3DRMFrame::SetVelocity

```
HRESULT SetVelocity(LPDIRECT3DRMFRAME lpRef, D3DVALUE rvX,
    D3DVALUE rvY, D3DVALUE rvZ, BOOL fRotVel);
```

Sets the velocity of the given frame relative to the reference frame. The frame will be moved by the vector [*rvX, rvY, rvZ*] with respect to the reference frame at each successive call to the IDirect3DRM::Tick or IDirect3DRMFrame::Move method.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpRef

Address of a variable that represents the Direct3DRMFrame object to be used as the reference.

rvX, rvY, and *rvZ*

New velocity for the frame.

fRotVel

Flag specifying whether the rotational velocity of the object is taken into account when setting the linear velocity. If TRUE, the object's rotational velocity is included in the calculation.

IDirect3DRMFrame::SetZbufferMode

```
HRESULT SetZbufferMode(D3DRMZBUFFERMODE d3drmZBM);
```

Sets the z-buffer mode; that is, whether z-buffering is enabled or disabled.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmZBM

One of the members of the D3DRMZBUFFERMODE enumerated type, specifying the z-buffer mode. The default value is D3DRMZBUFFER_FROMPARENT.

IDirect3DRMFrame::Transform

```
HRESULT Transform(D3DVECTOR *lpd3dVDst, D3DVECTOR *lpd3dVSrc);
```

Transforms the vector in the *lpd3dVSrc* parameter in frame coordinates to world coordinates, returning the result in the *lpd3dVDst* parameter.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpd3dVDst
Address of a D3DVECTOR structure that will be filled with the result of the transformation operation.

lpd3dVSrc
Address of a **D3DVECTOR** structure that is the source of the transformation operation.

IDirect3DRMLight Interface Method Groups

Applications use the methods of the **IDirect3DRMLight** interface to interact with light objects. The methods can be organized into the following groups:

Attenuation	GetConstantAttenuation GetLinearAttenuation GetQuadraticAttenuation SetConstantAttenuation SetLinearAttenuation SetQuadraticAttenuation
Color	GetColor SetColor SetColorRGB
Enable frames	GetEnableFrame SetEnableFrame
IUnknown	AddRef QueryInterface Release
Light types	GetType SetType
Range	GetRange SetRange
Spotlight options	GetPenumbra GetUmbra SetPenumbra SetUmbra

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMLight object without affecting the functionality of the original interface. In addition, the **IDirect3DRMLight** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMLight object is obtained by calling the IDirect3DRM::CreateLight or IDirect3DRM::CreateLightRGB method.

IDirect3DRMLight::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMLight object by 1. This method is part of the IUnknown interface inherited by Direct3DRMLight.

- Returns the new reference count of the object.

When the Direct3DRMLight object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMLight::Release method to decrease the reference count of the object by 1.

IDirect3DRMLight::GetColor

```
D3DCOLOR GetColor();
```

Retrieves the color of the current Direct3DRMLight object.

- Returns the color.

IDirect3DRMLight::GetConstantAttenuation

```
D3DVALUE GetConstantAttenuation();
```

Retrieves the constant attenuation factor for the Direct3DRMLight object.

- Returns the constant attenuation value.

The constant attenuation value affects the light intensity inversely. For example, a constant attenuation value of 2 halves the intensity of the light.

IDirect3DRMLight::GetEnableFrame

```
HRESULT GetEnableFrame(LPDIRECT3DRMFRAME * lplpEnableFrame);
```

Retrieves the enable frame for a light. The enable frame is the frame to which a light applies.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpEnableFrame
Address of a pointer that will contain the enable frame for the current Direct3DRMFrame object.

IDirect3DRMLight::GetLinearAttenuation

```
D3DVALUE GetLinearAttenuation();
```

Retrieves the linear attenuation factor for a light.

- Returns the linear attenuation value.

IDirect3DRMLight::GetPenumbra

```
D3DVALUE GetPenumbra();
```

Retrieves the penumbra angle of a spotlight.

- Returns the penumbra value.

IDirect3DRMLight::GetQuadraticAttenuation

```
D3DVALUE GetQuadraticAttenuation();
```

Retrieves the quadratic attenuation factor for a light.

- Returns the quadratic attenuation value.

IDirect3DRMLight::GetRange

```
D3DVALUE GetRange();
```

Retrieves the range of the current Direct3DRMLight object.

- Returns a value describing the range.

IDirect3DRMLight::GetType

```
D3DRMLIGHTTYPE GetType();
```

Retrieves the type of a given light.

- Returns one of the members of the D3DRMLIGHTTYPE enumerated type.

IDirect3DRMLight::GetUmbra

```
D3DVALUE GetUmbra();
```

Retrieves the umbra angle of the Direct3DRMLight object.

- Returns the umbra angle.

IDirect3DRMLight::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMLight object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMLight.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMLight::QueryInterface** method allows Direct3DRMLight objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMLight::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMLight object by 1. This method is part of the IUnknown interface inherited by Direct3DRMLight.

- Returns the new reference count of the object.

The Direct3DRMLight object deallocates itself when its reference count reaches 0. Use the IDirect3DRMLight::AddRef method to increase the reference count of the object by 1.

IDirect3DRMLight::SetColor

```
HRESULT SetColor(D3DCOLOR rcColor);
```

Sets the color of the given light.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rcColor
New color for the light.

IDirect3DRMLight::SetColorRGB

```
HRESULT SetColorRGB(D3DVALUE rvRed, D3DVALUE rvGreen,
    D3DVALUE rvBlue);
```

Sets the color of the given light.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvRed, rvGreen, and *rvBlue*
New color for the light.

IDirect3DRMLight::SetConstantAttenuation

```
HRESULT SetConstantAttenuation(D3DVALUE rvAtt);
```

Sets the constant attenuation factor for a light.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvAtt

New attenuation factor.

The constant attenuation value affects the light intensity inversely. For example, a constant attenuation value of 2 halves the intensity of the light.

IDirect3DRMLight::SetEnableFrame

```
HRESULT SetEnableFrame(LPDIRECT3DRMFRAME lpEnableFrame);
```

Sets the enable frame for a light. The enable frame is the frame to which a light applies.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpEnableFrame
Address of the light's enable frame. Child frames of this frame are also enabled for this light source.

IDirect3DRMLight::SetLinearAttenuation

```
HRESULT SetLinearAttenuation(D3DVALUE rvAtt);
```

Sets the linear attenuation factor for a light.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvAtt
New attenuation factor.

IDirect3DRMLight::SetPenumbra

```
HRESULT SetPenumbra(D3DVALUE rvAngle);
```

Sets the angle of the penumbra cone.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvAngle

New penumbra angle. This angle must be greater than or equal to the angle of the umbra. If you try to set the penumbra angle to less than the umbra angle, the umbra angle will be set equal to the penumbra angle. The default value is 0.5 radians.

IDirect3DRMLight::SetQuadraticAttenuation

```
HRESULT SetQuadraticAttenuation(D3DVALUE rvAtt);
```

Sets the quadratic attenuation factor for a light.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvAtt
New attenuation factor.

IDirect3DRMLight::SetRange

```
HRESULT SetRange(D3DVALUE rvRange);
```

Sets the range of a light. The light affects objects that are within the range only.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvRange
New range. The default value is 256.

IDirect3DRMLight::SetType

```
HRESULT SetType(D3DRMLIGHTTYPE d3drmtType);
```

Changes the light's type.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmtType
New light type specified by one of the members of the D3DRMLIGHTTYPE enumerated type.

IDirect3DRMLight::SetUmbra

```
HRESULT SetUmbra(D3DVALUE rvAngle);
```

Sets the angle of the umbra cone.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvAngle

New umbra angle. This angle must be less than or equal to the angle of the penumbra. If you try to set the umbra angle to greater than the penumbra angle, the penumbra angle will be set equal to the umbra angle. The default value is 0.4 radians.

IDirect3DRMMaterial Interface Method Groups

Applications use the methods of the **IDirect3DRMMaterial** interface to interact with material objects. The methods can be organized into the following groups:

Emission	GetEmissive SetEmissive
IUnknown	AddRef QueryInterface Release
Power for specular exponent	GetPower SetPower
Specular	GetSpecular SetSpecular

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMMaterial object without affecting the functionality of the original interface. In addition, the **IDirect3DRMMaterial** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMMaterial object is obtained by calling the IDirect3DRM::CreateMaterial method.

IDirect3DRMMaterial::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMMaterial object by 1. This method is part of the IUnknown interface inherited by Direct3DRMMaterial.

- Returns the new reference count of the object.

When the Direct3DRMMaterial object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMMaterial::Release method to decrease the reference count of the object by 1.

IDirect3DRMMaterial::GetEmissive

```
HRESULT GetEmissive(D3DVALUE *lpr, D3DVALUE *lpg, D3DVALUE *lpb);
```

Retrieves the setting for the emissive property of a material. The emissive setting is the color and intensity of the light the object emits.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpr, *lpg*, and *lpb*
Addresses that will contain the red, green, and blue components of the emissive color when the method returns.

IDirect3DRMMaterial::GetPower

```
D3DVALUE GetPower();
```

Retrieves the power used for the specular exponent in the given material.

- Returns the value specifying the power of the specular exponent.

IDirect3DRMMaterial::GetSpecular

```
HRESULT GetSpecular(D3DVALUE *lpr, D3DVALUE *lpg, D3DVALUE *lpb);
```

Retrieves the color of the specular highlights of a material.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpr, *lpg*, and *lpb*
Addresses that will contain the red, green, and blue components of the color of the specular highlights when the method returns.

IDirect3DRMMaterial::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMMaterial object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMMaterial.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMMaterial::QueryInterface** method allows Direct3DRMMaterial objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMMaterial::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMMaterial object by 1. This method is part of the IUnknown interface inherited by Direct3DRMMaterial.

- Returns the new reference count of the object.

The Direct3DRMMaterial object deallocates itself when its reference count reaches 0. Use the IDirect3DRMMaterial::AddRef method to increase the reference count of the object by 1.

IDirect3DRMMaterial::SetEmissive

```
HRESULT SetEmissive(D3DVALUE r, D3DVALUE g, D3DVALUE b);
```

Sets the emissive property of a material. The emissive property is the color and intensity of the light the object emits.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

r, *g*, and *b*
Red, green, and blue components of the emissive color.

IDirect3DRMMaterial::SetPower

```
HRESULT SetPower(D3DVALUE rvPower);
```

Sets the power used for the specular exponent in a material.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvPower
New specular exponent.

IDirect3DRMMaterial::SetSpecular

```
HRESULT SetSpecular(D3DVALUE r, D3DVALUE g, D3DVALUE b);
```

Sets the color of the specular highlights for a material.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

r, *g*, and *b*
Red, green, and blue components of the color of the specular highlights.

IDirect3DRMMesh Interface Method Groups

Applications use the methods of the **IDirect3DRMMesh** interface to interact with groups of meshes. The methods can be organized into the following groups:

Color	GetGroupColor SetGroupColor SetGroupColorRGB
Creation and information	AddGroup GetBox GetGroup GetGroupCount
IUnknown	AddRef QueryInterface Release
Materials	GetGroupMaterial SetGroupMaterial
Miscellaneous	Scale Translate
Rendering quality	GetGroupQuality SetGroupQuality
Texture mapping	GetGroupMapping SetGroupMapping
Textures	GetGroupTexture SetGroupTexture
Vertex positions	GetVertices SetVertices

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMMesh object without affecting the functionality of the original interface. In addition, the **IDirect3DRMMesh** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMMesh object is obtained by calling the IDirect3DRM::CreateMesh method.

IDirect3DRMMesh::AddGroup

```
HRESULT AddGroup(unsigned vCount, unsigned fCount,
    unsigned vPerFace, unsigned *fData, D3DRMGROUPINDEX *returnId);
```

Groups a collection of faces and retrieves an identifier for the group.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

vCount and *fCount*
Number of vertices and faces in the group.

vPerFace
Number of vertices per face in the group, if all faces have the same vertex count. If the group contains faces with varying vertex counts, this parameter should be zero.

fData
Address of face data. If the *vPerFace* parameter specifies a value, this data is simply a list of indices into the group's vertex array. If *vPerFace* is zero, the vertex indices should be preceded by an integer giving the number of vertices in that face. For example, if *vPerFace* is zero and the group is made up of triangular and quadrilateral faces, the data might be in the following form: 3 *index index index* 4 *index index index index* 3 *index index index* ...

returnId
Address of a variable that will identify the group when the method returns.

A newly added group has the following default properties:

- White
- No texture
- No specular reflection
- Position, normal, and color of each vertex in the vertex array equal to zero

To set the positions of the vertices, use the IDirect3DRMMesh::SetVertices method.

IDirect3DRMMesh::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMMesh object by 1. This method is part of the IUnknown interface inherited by Direct3DRMMesh.

- Returns the new reference count of the object.

When the Direct3DRMMesh object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMMesh::Release method to decrease the reference count of the object by 1.

IDirect3DRMMesh::GetBox

```
HRESULT GetBox(D3DRMBOX * lpD3DRMBox);
```

Retrieves the bounding box containing a Direct3DRMMesh object. The bounding box gives the minimum and maximum model coordinates in each dimension.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

IDirect3DRMMesh::GetGroup

```
HRESULT GetGroup(D3DRMGROUPINDEX id, unsigned *vCount,
    unsigned *fCount, unsigned *vPerFace, DWORD *fDataSize,
    unsigned *fData);
```

Retrieves the data associated with a specified group.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

vCount and *fCount*

Addresses of variables that will contain the number of vertices and the number of faces for the group when the method returns. These parameters can be NULL.

vPerFace

Address of a variable that will contain the number of vertices per face for the group when the method returns. This parameter can be NULL.

fDataSize

Address of a variable that specifies the number of unsigned elements in the buffer pointed to by the *fData* parameter. This parameter cannot be NULL.

fData

Address of a buffer that will contain the face data for the group when the method returns. The format of this data is the same as was specified in the call to the IDirect3DRMMesh::AddGroup method. If this parameter is NULL, the method returns the required size of the buffer in the *fDataSize* parameter.

IDirect3DRMMesh::GetGroupColor

```
D3DCOLOR GetGroupColor(D3DRMGROUPINDEX id);
```

Retrieves the color for a group.

- Returns a D3DCOLOR variable specifying the color if successful, or zero otherwise.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

IDirect3DRMMesh::GetGroupCount

```
unsigned GetGroupCount();
```

Retrieves the number of groups for a given Direct3DRMMesh object.

- Returns the number of groups if successful, or zero otherwise.

IDirect3DRMMesh::GetGroupMapping

```
D3DRMMAPPING GetGroupMapping(D3DRMGROUPINDEX id);
```

Returns a description of how textures are mapped to a group in a Direct3DRMMesh object.

- Returns one of the D3DRMMAPPING values describing how textures are mapped to a group, if successful. Returns zero otherwise.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

IDirect3DRMMesh::GetGroupMaterial

```
HRESULT GetGroupMaterial(D3DRMGROUPINDEX id,
    LPDIRECT3DRMMATERIAL *returnPtr);
```

Retrieves a pointer to the material associated with a group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

returnPtr

Address of a pointer to a variable that will contain the IDirect3DRMMaterial interface for the group when the method returns.

IDirect3DRMMesh::GetGroupQuality

```
D3DRMRENDERQUALITY GetGroupQuality(D3DRMGROUPINDEX id);
```

Retrieves the rendering quality for a specified group in a Direct3DRMMesh object.

- Returns values from the enumerated types represented by D3DRMRENDERQUALITY if successful, or zero otherwise. These values include the shading, lighting, and fill modes for the object.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

IDirect3DRMMesh::GetGroupTexture

```
HRESULT GetGroupTexture(D3DRMGROUPINDEX id,
    LPDIRECT3DRMTEXTURE *returnPtr);
```

Retrieves an address of the texture associated with a group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

returnPtr

Address of a pointer to a variable that will contain the IDirect3DRMTexture interface for the group when the method returns.

IDirect3DRMMesh::GetVertices

```
HRESULT GetVertices(D3DRMGROUPINDEX id, DWORD index,
    DWORD count, D3DRMVERTEX *returnPtr);
```

Retrieves the vertex positions for a specified group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

index

Index into the array of D3DRMVERTEX structures at which to begin returning vertex positions.

count

Number of **D3DRMVERTEX** structures (vertices) to retrieve following the index given in the *index* parameter. This parameter cannot be NULL.

returnPtr

Array of **D3DRMVERTEX** structures that will contain the vertex positions when the method returns. If this parameter is NULL, the method returns the required number of **D3DRMVERTEX** structures in the *count* parameter.

IDirect3DRMMesh::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMMesh object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMMesh.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMMesh::QueryInterface** method allows Direct3DRMMesh objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMMesh::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMMesh object by 1. This method is part of the IUnknown interface inherited by Direct3DRMMesh.

- Returns the new reference count of the object.

The Direct3DRMMesh object deallocates itself when its reference count reaches 0. Use the IDirect3DRMMesh::AddRef method to increase the reference count of the object by 1.

IDirect3DRMMesh::Scale

```
HRESULT Scale(D3DVALUE sx, D3DVALUE sy, D3DVALUE sz);
```

Scales a Direct3DRMMesh object by the given scaling factors, parallel to the x-, y-, and z-axes in model coordinates.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

sx, sy, and *sz*
Scaling factors that are applied along the x-, y-, and z-axes.

IDirect3DRMMesh::SetGroupColor

```
HRESULT SetGroupColor(D3DRMGROUPINDEX id, D3DCOLOR value);
```

Sets the color of a group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

value

Color of the group.

IDirect3DRMMesh::SetGroupColorRGB

```
HRESULT SetGroupColorRGB(D3DRMGROUPINDEX id, D3DVALUE red,
    D3DVALUE green, D3DVALUE blue);
```

Sets the color of a group in a Direct3DRMMesh object, using individual RGB values.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

red, *green*, and *blue*

Red, green, and blue components of the group color.

IDirect3DRMMesh::SetGroupMapping

```
HRESULT SetGroupMapping(D3DRMGROUPINDEX id, D3DRMMAPPING value);
```

Sets the mapping for a group in a Direct3DRMMesh object. The mapping controls how textures are mapped to a surface.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

value

Value of the D3DRMMAPPING type describing the mapping for the group.

IDirect3DRMMesh::SetGroupMaterial

```
HRESULT SetGroupMaterial(D3DRMGROUPINDEX id, LPDIRECT3DRMMATERIAL value);
```

Sets the material associated with a group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

value

Address of the IDirect3DRMMaterial interface for the Direct3DRMMesh object.

IDirect3DRMMesh::SetGroupQuality

```
HRESULT SetGroupQuality(D3DRMGROUPINDEX id, D3DRMRENDERQUALITY value);
```

Sets the rendering quality for a specified group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

value

Values from the enumerated types represented by the D3DRMRENDERQUALITY type. These values include the shading, lighting, and fill modes for the object.

IDirect3DRMMesh::SetGroupTexture

```
HRESULT SetGroupTexture(D3DRMGROUPINDEX id, LPDIRECT3DRMTEXTURE value);
```

Sets the texture associated with a group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

value

Address of the IDirect3DRMTexture interface for the Direct3DRMMesh object.

IDirect3DRMMesh::SetVertices

```
HRESULT SetVertices(D3DRMGROUPINDEX id, unsigned index,
    unsigned count, D3DRMVERTEX *values);
```

Sets the vertex positions for a specified group in a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

index

Index into the array specified in the *values* parameter at which to begin setting vertex positions.

count

Number of vertices to set following the index given in the *index* parameter.

values

Array of D3DRMVERTEX structures specifying the vertex positions to be set.

IDirect3DRMMesh::Translate

```
HRESULT Translate(D3DVALUE tx, D3DVALUE ty, D3DVALUE tz);
```

Adds the specified offsets to the vertex positions of a Direct3DRMMesh object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

tx, ty, and *tz*
Offsets that are added to the x-, y-, and z-coordinates respectively of each vertex position.

IDirect3DRMMeshBuilder Interface Method Groups

Applications use the methods of the **IDirect3DRMMeshBuilder** interface to interact with mesh objects. The methods can be organized into the following groups:

Color	GetColorSource SetColor SetColorRGB SetColorSource
Creation and information	GetBox
Faces	AddFace AddFaces CreateFace GetFaceCount GetFaces
IUnknown	AddRef QueryInterface Release
Loading	Load
Meshes	AddMesh CreateMesh
Miscellaneous	AddFrame AddMeshBuilder ReserveSpace Save Scale SetMaterial Translate
Normals	AddNormal GenerateNormals SetNormal
Perspective	GetPerspective SetPerspective
Rendering quality	GetQuality SetQuality
Textures	GetTextureCoordinates SetTexture SetTextureCoordinates

	SetTextureTopology
Vertices	AddVertex
	GetVertexColor
	GetVertexCount
	GetVertices
	SetVertex
	SetVertexColor
	SetVertexColorRGB

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMMeshBuilder object without affecting the functionality of the original interface. In addition, the **IDirect3DRMMeshBuilder** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMMeshBuilder object is obtained by calling the IDirect3DRM::CreateMeshBuilder method.

IDirect3DRMMeshBuilder::AddFace

```
HRESULT AddFace(LPDIRECT3DRMFACE lpD3DRMFace);
```

Adds a face to a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMFace
Address of the face being added.

Any one face can exist in only one mesh at a time.

IDirect3DRMMeshBuilder::AddFaces

```
HRESULT AddFaces(DWORD dwVertexCount, D3DVECTOR * lpD3DVertices,
    DWORD normalCount, D3DVECTOR *lpNormals, DWORD *lpFaceData,
    LPDIRECT3DRMFACEARRAY* lplpD3DRMFaceArray);
```

Adds faces to a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

dwVertexCount
Number of vertices.

lpD3DVertices
Base address of an array of D3DVECTOR structures that stores the vertex positions.

normalCount
Number of normals.

lpNormals
Base address of an array of **D3DVECTOR** structures that stores the normal positions.

lpFaceData
For each face, this parameter should contain a vertex count followed by the indices into the vertices array. If *normalCount* is not zero, this parameter should contain a vertex count followed by pairs of indices, with the first index of each pair indexing into the array of vertices, and the second indexing into the array of normals. The list of indices must terminate with a zero.

lplpD3DRMFaceArray
Address of a pointer to an IDirect3DRMFaceArray interface that will be filled with a pointer to the newly created faces.

IDirect3DRMMeshBuilder::AddFrame

```
HRESULT AddFrame(LPDIRECT3DRMFRAME lpD3DRMFrame);
```

Adds the contents of a frame to a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMFrame
Address of the frame whose contents are being added.

The source frame is not modified or referenced by this operation.

IDirect3DRMMeshBuilder::AddMesh

```
HRESULT AddMesh(LPDIRECT3DRMMESH lpD3DRMMesh);
```

Adds a mesh to a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMMesh
Address of the mesh being added.

IDirect3DRMMeshBuilder::AddMeshBuilder

```
HRESULT AddMeshBuilder(LPDIRECT3DRMMESHBUILDER lpD3DRMMeshBuild);
```

Adds the contents of a Direct3DRMMeshBuilder object to another Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMMeshBuild
Address of the Direct3DRMMeshBuilder object whose contents are being added.

The source Direct3DRMMeshBuilder object is not modified or referenced by this operation.

IDirect3DRMMeshBuilder::AddNormal

```
int AddNormal(D3DVALUE x, D3DVALUE y, D3DVALUE z);
```

Adds a normal to a Direct3DRMMeshBuilder object.

- Returns the index of the normal.

x, y, and *z*
The x, y, and z components of the direction of the new normal.

IDirect3DRMMeshBuilder::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMMeshBuilder object by 1. This method is part of the IUnknown interface inherited by Direct3DRMMeshBuilder.

- Returns the new reference count of the object.

When the Direct3DRMMeshBuilder object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMMeshBuilder::Release method to decrease the reference count of the object by 1.

IDirect3DRMMeshBuilder::AddVertex

```
int AddVertex(D3DVALUE x, D3DVALUE y, D3DVALUE z);
```

Adds a vertex to a Direct3DRMMeshBuilder object.

- Returns the index of the vertex.

x, y, and *z*
The x, y, and z components of the position of the new vertex.

IDirect3DRMMeshBuilder::CreateFace

```
HRESULT CreateFace(LPDIRECT3DRMFACE* lplpD3DRMFace);
```

Creates a new face with no vertices and adds it to a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRMFace
Address of a pointer to an IDirect3DRMFace interface that will be filled with a pointer to the face that was created.

IDirect3DRMMeshBuilder::CreateMesh

```
HRESULT CreateMesh(LPDIRECT3DRMMESH* lplpD3DRMMesh);
```

Creates a new mesh from a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRMMesh
Address that will be filled with a pointer to an IDirect3DRMMesh interface.

IDirect3DRMMeshBuilder::GenerateNormals

```
HRESULT GenerateNormals();
```

Processes the Direct3DRMMeshBuilder object and generates vertex normals that are the average of each vertex's adjoining face normals.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Averaging the normals of back-to-back faces produces a zero normal.

IDirect3DRMMeshBuilder::GetBox

```
HRESULT GetBox(D3DRMBOX *lpD3DRMBox);
```

Retrieves the bounding box containing a Direct3DRMMeshBuilder object. The bounding box gives the minimum and maximum model coordinates in each dimension.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

IDirect3DRMMeshBuilder::GetColorSource

```
D3DRMCOLORSOURCE GetColorSource();
```

Retrieves the color source of a Direct3DRMMeshBuilder object. The color source can be either a face or a vertex.

- Returns a member of the D3DRMCOLORSOURCE enumerated type.

IDirect3DRMMeshBuilder::GetFaceCount

```
int GetFaceCount();
```

Retrieves the number of faces in a Direct3DRMMeshBuilder object.

- Returns the number of faces.

IDirect3DRMMeshBuilder::GetFaces

```
HRESULT GetFaces(LPDIRECT3DRMFACEARRAY* lplpD3DRMFaceArray);
```

Retrieves the faces of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DRMFaceArray
Address of a pointer to an IDirect3DRMFaceArray interface that is filled with an address of the faces.

IDirect3DRMMeshBuilder::GetPerspective

```
BOOL GetPerspective();
```

Determines whether perspective correction is on for a Direct3DRMMeshBuilder object.

- Returns TRUE if perspective correction is on, or FALSE otherwise.

IDirect3DRMMeshBuilder::GetQuality

```
D3DRMRENDERQUALITY GetQuality();
```

Retrieves the rendering quality of a Direct3DRMMeshBuilder object.

- Returns a member of the D3DRMRENDERQUALITY enumerated type that specifies the rendering quality of the mesh.

IDirect3DRMMeshBuilder::GetTextureCoordinates

```
HRESULT GetTextureCoordinates(DWORD index, D3DVALUE *lpU,
    D3DVALUE *lpV);
```

Retrieves the texture coordinates of a specified vertex in a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the vertex.

lpU and *lpV*
Addresses of variables that will be filled with the texture coordinates of the vertex when the method returns.

IDirect3DRMMeshBuilder::GetVertexColor

```
D3DCOLOR GetVertexColor(DWORD index);
```

Retrieves the color of a specified vertex in a Direct3DRMMeshBuilder object.

- Returns the color.

index
Index of the vertex.

IDirect3DRMMeshBuilder::GetVertexCount

```
int GetVertexCount();
```

Retrieves the number of vertices in a Direct3DRMMeshBuilder object.

- Returns the number of vertices.

IDirect3DRMMeshBuilder::GetVertices

```
HRESULT GetVertices(DWORD *vcount, D3DVECTOR *vertices,
    DWORD *ncount, D3DVECTOR *normals, DWORD *face_data_size,
    DWORD *face_data);
```

Retrieves the vertices, normals, and face data for a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

vcount
Address of a variable that will contain the number of vertices.

vertices
Address of an array of D3DVECTOR structures that will contain the vertices for the Direct3DRMMeshBuilder object.

ncount
Address of a variable that will contain the number of normals.

normals
Array of **D3DVECTOR** structures that will contain the normals for the Direct3DRMMeshBuilder object.

face_data_size
Address of a variable that specifies the size of the buffer pointed to by the *face_data* parameter. The size is given in units of **DWORD** values. This parameter cannot be NULL.

face_data
Address of the face data for the Direct3DRMMeshBuilder object. This data is in the same format as specified in the IDirect3DRMMesh::AddGroup method except that it is null-terminated. If this parameter is NULL, the method returns the required size of the face-data buffer in the *face_data_size* parameter.

IDirect3DRMMeshBuilder::Load

```
HRESULT Load(LPVOID lpvObjSource, LPVOID lpvObjID,
    D3DRMLOADOPTIONS d3drmLOFlags,
    D3DRMLOADTEXTURECALLBACK d3drmLoadTextureProc, LPVOID lpvArg);
```

Loads a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpvObjSource

Source for the object to be loaded. This source can be a file, resource, memory block, or stream, depending on the source flags specified in the *d3drmLOFlags* parameter.

lpvObjID

Object name or position to be loaded. The use of this parameter depends on the identifier flags specified in the *d3drmLOFlags* parameter. If the D3DRMLOAD_BYPOSITION flag is specified, this parameter is a pointer to a **DWORD** value that gives the object's order in the file. This parameter can be NULL.

d3drmLOFlags

Value of the D3DRMLOADOPTIONS type describing the load options.

d3drmLoadTextureProc

A D3DRMLOADTEXTURECALLBACK callback function called to load any textures used by an object.

lpvArg

Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

By default, this method loads the first mesh from the source specified in the *lpvObjSource* parameter.

IDirect3DRMMeshBuilder::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMMeshBuilder object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMMeshBuilder.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMMeshBuilder::QueryInterface** method allows Direct3DRMMeshBuilder objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMMeshBuilder::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMMeshBuilder object by 1. This method is part of the IUnknown interface inherited by Direct3DRMMeshBuilder.

- Returns the new reference count of the object.

The Direct3DRMMeshBuilder object deallocates itself when its reference count reaches 0. Use the IDirect3DRMMeshBuilder::AddRef method to increase the reference count of the object by 1.

IDirect3DRMMeshBuilder::ReserveSpace

```
HRESULT ReserveSpace(DWORD vertexCount, DWORD normalCount,
    DWORD faceCount);
```

Reserves space within a Direct3DRMMeshBuilder object for the specified number of vertices, normals, and faces. This allows the system to use memory more efficiently.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

vertexCount, normalCount, and *faceCount*
Number of vertices, normals, and faces to allocate space for.

IDirect3DRMMeshBuilder::Save

```
HRESULT Save(const char * lpFilename,
    D3DRMXOFFORMAT d3drmXOFFormat, D3DRMSAVEOPTIONS d3drmSOContents);
```

Saves a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpFilename
Address specifying the name of the created file. This file must have a .X file name extension.

d3drmXOFFormat
The D3DRMXOF_TEXT value from the D3DRMXOFFORMAT enumerated type.

d3drmSOContents
Value of the D3DRMSAVEOPTIONS type describing the save options.

IDirect3DRMMeshBuilder::Scale

```
HRESULT Scale(D3DVALUE sx, D3DVALUE sy, D3DVALUE sz);
```

Scales a Direct3DRMMeshBuilder object by the given scaling factors, parallel to the x-, y-, and z-axes in model coordinates.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

sx, sy, and *sz*
Scaling factors that are applied along the x-, y-, and z-axes.

IDirect3DRMMeshBuilder::SetColor

```
HRESULT SetColor(D3DCOLOR color);
```

Sets all the faces of a Direct3DRMMeshBuilder object to a given color.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

color
Color of the faces.

IDirect3DRMMeshBuilder::SetColorRGB

```
HRESULT SetColorRGB(D3DVALUE red, D3DVALUE green, D3DVALUE blue);
```

Sets all the faces of a Direct3DRMMeshBuilder object to a given color.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

red, green, and *blue*
Red, green, and blue components of the color.

IDirect3DRMMeshBuilder::SetColorSource

```
HRESULT SetColorSource(D3DRMCOLORSOURCE source);
```

Sets the color source of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

source
Member of the D3DRMCOLORSOURCE enumerated type that specifies the new color source to use.

IDirect3DRMMeshBuilder::SetMaterial

```
HRESULT SetMaterial(LPDIRECT3DRMMATERIAL lpIDirect3DRMmaterial);
```

Sets the material of all the faces of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpIDirect3DRMmaterial
Address of IDirect3DRMMaterial interface for the Direct3DRMMeshBuilder object.

IDirect3DRMMeshBuilder::SetNormal

```
HRESULT SetNormal(DWORD index, D3DVALUE x, D3DVALUE y, D3DVALUE z);
```

Sets the normal vector of a specified vertex in a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the normal to be set.

x, y, and *z*
The x, y, and z components of the vector to assign to the specified normal.

IDirect3DRMMeshBuilder::SetPerspective

```
HRESULT SetPerspective(BOOL perspective);
```

Enables or disables perspective-correct texture-mapping for a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

perspective
Specify TRUE if the mesh should be texture-mapped with perspective correction, or FALSE otherwise.

IDirect3DRMMeshBuilder::SetQuality

```
HRESULT SetQuality(D3DRMRENDERQUALITY quality);
```

Sets the rendering quality of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

quality
Member of the D3DRMRENDERQUALITY enumerated type that specifies the new rendering quality to use.

IDirect3DRMMeshBuilder::SetTexture

```
HRESULT SetTexture(LPDIRECT3DRMTEXTURE lpD3DRMTexture);
```

Sets the texture of all the faces of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMTexture
Address of the required Direct3DRMTexture object.

IDirect3DRMMeshBuilder::SetTextureCoordinates

```
HRESULT SetTextureCoordinates(DWORD index, D3DVALUE u, D3DVALUE v);
```

Sets the texture coordinates of a specified vertex in a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the vertex to be set.

u and *v*
Texture coordinates to assign to the specified mesh vertex.

IDirect3DRMMeshBuilder::SetTextureTopology

```
HRESULT SetTextureTopology(BOOL cylU, BOOL cylV);
```

Sets the texture topology of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

cylU and *cylV*
Specify TRUE for either or both of these parameters if you want the texture to have a cylindrical topology in the u and v dimensions respectively; otherwise, specify FALSE.

IDirect3DRMMeshBuilder::SetVertex

```
HRESULT SetVertex(DWORD index, D3DVALUE x, D3DVALUE y, D3DVALUE z);
```

Sets the position of a specified vertex in a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the vertex to be set.

x, y, and *z*
The x, y, and z components of the position to assign to the specified vertex.

IDirect3DRMMeshBuilder::SetVertexColor

```
HRESULT SetVertexColor(DWORD index, D3DCOLOR color);
```

Sets the color of a specified vertex in a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the vertex to be set.

color
Color to assign to the specified vertex.

IDirect3DRMMeshBuilder::SetVertexColorRGB

```
HRESULT SetVertexColorRGB(DWORD index, D3DVALUE red,
    D3DVALUE green, D3DVALUE blue);
```

Sets the color of a specified vertex in a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

index
Index of the vertex to be set.

red, green, and *blue*
Red, green, and blue components of the color to assign to the specified vertex.

IDirect3DRMMeshBuilder::Translate

```
HRESULT Translate(D3DVALUE tx, D3DVALUE ty, D3DVALUE tz);
```

Adds the specified offsets to the vertex positions of a Direct3DRMMeshBuilder object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

tx, ty, and *tz*
Offsets that are added to the x-, y-, and z-coordinates respectively of each vertex position.

IDirect3DRMObject Interface Method Groups

Applications use the methods of the **IDirect3DRMObject** interface to work with the object superclass of Direct3DRM objects. The methods can be organized into the following groups:

Application-specific data	GetAppData SetAppData
Cloning	Clone
IUnknown	AddRef QueryInterface Release
Naming	GetClassName GetName SetName
Notifications	AddDestroyCallback DeleteDestroyCallback

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMObject object without affecting the functionality of the original interface.

The Direct3DRMObject object is obtained through the appropriate call to the **QueryInterface** method from any Direct3DRM object. All Direct3DRM objects inherit the **IDirect3DRMObject** interface methods.

IDirect3DRMObject::AddDestroyCallback

```
HRESULT AddDestroyCallback(D3DRMOBJECTCALLBACK lpCallback,
    LPVOID lpArg);
```

Registers a function to be called when an object is destroyed.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpCallback

User-defined callback function that will be called when the object is destroyed.

lpArg

Address of application-defined data passed to the callback function. Because this function is called after the object has been destroyed, you should not call this function with the object as an argument.

IDirect3DRMObject::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMObject object by 1. This method is part of the IUnknown interface inherited by Direct3DRMObject.

- Returns the new reference count of the object.

When the Direct3DRMObject object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMObject::Release method to decrease the reference count of the object by 1.

IDirect3DRMObject::Clone

```
HRESULT Clone(LPUNKNOWN pUnkOuter, REFIID riid, LPVOID *ppvObj);
```

Creates a copy of an object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

pUnkOuter
Allows COM aggregation features.

riid
Identifier of the object being copied.

ppvObj
Address that will contain the copy of the object when the method returns.

IDirect3DRMObject::DeleteDestroyCallback

```
HRESULT DeleteDestroyCallback(D3DRMOBJECTCALLBACK d3drmObjProc,
    LPVOID lpArg);
```

Removes a function previously registered with the IDirect3DRMObject::AddDestroyCallback method.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmObjProc

User-defined D3DRMOBJECTCALLBACK callback function that will be called when the object is destroyed.

lpArg

Address of application-defined data passed to the callback function.

IDirect3DRMObject::GetAppData

```
DWORD GetAppData();
```

Retrieves the 32 bits of application-specific data in the object. The default value is 0.

- Returns the data value defined by the application.

IDirect3DRMObject::GetClassName

```
HRESULT GetClassName(LPDWORD lpdwSize, LPSTR lpName);
```

Retrieves the name of the object's class.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpdwSize

Address of a variable containing the size, in bytes, of the buffer pointed to by the *lpName* parameter.

lpName

Address of a variable that will contain a null-terminated string identifying the class name when the method returns. If this parameter is NULL, the *lpdwSize* parameter will contain the required size for the string when the method returns.

IDirect3DRMObject::GetName

```
HRESULT GetName(LPDWORD lpdwSize, LPSTR lpName);
```

Retrieves the object's name.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpdwSize

Address of a variable containing the size, in bytes, of the buffer pointed to by the *lpName* parameter.

lpName

Address of a variable that will contain a null-terminated string identifying the object's name when the method returns. If this parameter is NULL, the *lpdwSize* parameter will contain the required size for the string when the method returns.

IDirect3DRMObject::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMObject object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMObject.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMObject::QueryInterface** method allows Direct3DRMObject objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMObject::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMObject object by 1. This method is part of the IUnknown interface inherited by Direct3DRMObject.

- Returns the new reference count of the object.

The Direct3DRMObject object deallocates itself when its reference count reaches 0. Use the IDirect3DRMObject::AddRef method to increase the reference count of the object by 1.

IDirect3DRMObject::SetAppData

```
HRESULT SetAppData(DWORD ulData);
```

Sets the 32 bits of application-specific data in the object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

ulData
User-defined data to be stored with the object.

IDirect3DRMObject::SetName

```
HRESULT SetName(const char * lpName);
```

Sets the object's name.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpName
User-defined data to be the name for the object.

IDirect3DRMShadow Interface Method Groups

Applications use the **IDirect3DRMShadow** interface to initialize Direct3DRMShadow objects. Note that this initialization is not necessary if the application calls the IDirect3DRM::CreateShadow method; it is required only if the application calls the IDirect3DRM::CreateObject method to create the shadow. This interface supports the following methods:

Initialization	Init
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMShadow object without affecting the functionality of the original interface. In addition, the **IDirect3DRMShadow** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMShadow object is obtained by calling the IDirect3DRM::CreateShadow method.

IDirect3DRMShadow::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMShadow object by 1. This method is part of the IUnknown interface inherited by Direct3DRMShadow.

- Returns the new reference count of the object.

When the Direct3DRMShadow object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMShadow::Release method to decrease the reference count of the object by 1.

IDirect3DRMShadow::Init

```
HRESULT Init(LPDIRECT3DRMVISUAL lpD3DRMVisual,
    LPDIRECT3DRMLIGHT lpD3DRMLight, D3DVALUE px, D3DVALUE py,
    D3DVALUE pz, D3DVALUE nx, D3DVALUE ny, D3DVALUE nz);
```

Initializes a Direct3DRMShadow object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMVisual
Address of the Direct3DRMVisual object casting the shadow.

lpD3DRMLight
Address of the Direct3DRMLight object that provides the light that defines the shadow.

px, *py*, and *pz*
Coordinates of a point on the plane on which the shadow is cast.

nx, *ny*, and *nz*
Coordinates of the normal vector of the plane on which the shadow is cast.

IDirect3DRMShadow::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMShadow object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMShadow.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMShadow::QueryInterface** method allows Direct3DRMShadow objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMShadow::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMShadow object by 1. This method is part of the IUnknown interface inherited by Direct3DRMShadow.

- Returns the new reference count of the object.

The Direct3DRMShadow object deallocates itself when its reference count reaches 0. Use the IDirect3DRMShadow::AddRef method to increase the reference count of the object by 1.

IDirect3DRMTexture Interface Method Groups

Applications use the methods of the **IDirect3DRMTexture** interface to work with textures, which are rectangular arrays of pixels. The methods can be organized into the following groups:

Color	GetColors SetColors
Decals	GetDecalOrigin GetDecalScale GetDecalSize GetDecalTransparency GetDecalTransparentColor SetDecalOrigin SetDecalScale SetDecalSize SetDecalTransparency SetDecalTransparentColor
Images	GetImage
Initialization	InitFromFile InitFromResource InitFromSurface
IUnknown	AddRef QueryInterface Release
Renderer notification	Changed
Shading	GetShades SetShades

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMTexture object without affecting the functionality of the original interface. In addition, the **IDirect3DRMTexture** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMTexture object is obtained by calling the IDirect3DRM::CreateTexture method.

IDirect3DRMTexture::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMTexture object by 1. This method is part of the IUnknown interface inherited by Direct3DRMTexture.

- Returns the new reference count of the object.

When the Direct3DRMTexture object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMTexture::Release method to decrease the reference count of the object by 1.

IDirect3DRMTexture::Changed

```
HRESULT Changed(BOOL bPixels, BOOL bPalette);
```

Informs the renderer that the application has changed the pixels or the palette of a texture.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

bPixels
If this parameter is TRUE, the pixels have changed.

bPalette
If this parameter is TRUE, the palette has changed.

IDirect3DRMTexture::GetColors

```
DWORD GetColors();
```

Retrieves the maximum number of colors used for rendering a texture.

- Returns the number of colors.

This method returns the number of colors that the texture has been quantized to, not the number of colors in the image from which the texture was created. Consequently, the number of colors that are returned usually matches the colors that were set by calling the IDirect3DRM::SetDefaultTextureColors method, unless you used the IDirect3DRMTexture::SetColors method explicitly to change the colors for the texture.

IDirect3DRMTexture::GetDecalOrigin

```
HRESULT GetDecalOrigin(LONG * lplX, LONG * lplY);
```

Retrieves the current origin of the decal.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplX and *lplY*
Addresses of variables that will be filled with the origin of the decal when the method returns.

IDirect3DRMTexture::GetDecalScale

```
DWORD GetDecalScale();
```

Retrieves the scaling property of the given decal.

- Returns the scaling property if successful, or -1 otherwise.

IDirect3DRMTexture::GetDecalSize

```
HRESULT GetDecalSize(D3DVALUE *lprvWidth, D3DVALUE *lprvHeight);
```

Retrieves the size of the decal.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lprvWidth and *lprvHeight*
Addresses of variables that will be filled with the width and height of the decal when the method returns.

IDirect3DRMTexture::GetDecalTransparency

```
BOOL GetDecalTransparency();
```

Retrieves the transparency property of the decal.

- Returns TRUE if the decal has a transparent color, FALSE otherwise.

IDirect3DRMTexture::GetDecalTransparentColor

```
D3DCOLOR GetDecalTransparentColor();
```

Retrieves the transparent color of the decal.

- Returns the value of the transparent color.

IDirect3DRMTexture::GetImage

```
D3DRMIMAGE * GetImage();
```

Returns an address of the image that the texture was created with.

- Returns the address of the D3DRMIMAGE structure that the current texture was created with.

IDirect3DRMTexture::GetShades

```
DWORD GetShades();
```

Retrieves the number of shades used for each color in the texture when rendering.

- Returns the number of shades.

IDirect3DRMTexture::InitFromFile

```
HRESULT InitFromFile(const char *filename);
```

Initializes a texture by using the information in a given file.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

filename
Address of a string specifying the file from which initialization information is drawn.

IDirect3DRMTexture::InitFromResource

```
HRESULT InitFromResource(HRSRC rs);
```

Initializes a Direct3DRMTexture object from a specified resource.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rs
Handle of the specified resource.

IDirect3DRMTexture::InitFromSurface

```
HRESULT InitFromSurface(LPDIRECTDRAWSURFACE lpDDS);
```

Initializes a texture by using the data from a given DirectDraw surface.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpDDS
Address of a DirectDraw surface from which initialization information is drawn.

IDirect3DRMTexture::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMTexture object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMTexture.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMTexture::QueryInterface** method allows Direct3DRMTexture objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMTexture::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMTexture object by 1. This method is part of the IUnknown interface inherited by Direct3DRMTexture.

- Returns the new reference count of the object.

The Direct3DRMTexture object deallocates itself when its reference count reaches 0. Use the IDirect3DRMTexture::AddRef method to increase the reference count of the object by 1.

IDirect3DRMTexture::SetColors

```
HRESULT SetColors(DWORD ulColors);
```

Sets the maximum number of colors used for rendering a texture. This method is required only in the ramp color model.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

ulColors
Number of colors. The default value is 8.

IDirect3DRMTexture::SetDecalOrigin

```
HRESULT SetDecalOrigin(LONG lX, LONG lY);
```

Sets the origin of the decal as an offset from the top left of the decal.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lX and *lY*

New origin, in decal coordinates, for the decal. The default origin is [0, 0].

The decal's origin is mapped to its frame's position when rendering. For example, the origin of a decal of a cross would be set to the middle of the decal, and the origin of an arrow pointing down would be set to midway along the bottom edge.

IDirect3DRMTexture::SetDecalScale

```
HRESULT SetDecalScale(DWORD dwScale);
```

Sets the scaling property for a decal.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

dwScale

If this parameter is TRUE, depth is taken into account when the decal is scaled. If it is FALSE, depth information is ignored. The default value is TRUE.

IDirect3DRMTexture::SetDecalSize

```
HRESULT SetDecalSize(D3DVALUE rvWidth, D3DVALUE rvHeight);
```

Sets the size of the decal to be used if the decal is being scaled according to its depth in the scene.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvWidth and *rvHeight*
New width and height, in model coordinates, of the decal. The default size is [1, 1].

IDirect3DRMTexture::SetDecalTransparency

```
HRESULT SetDecalTransparency(BOOL bTransp);
```

Sets the transparency property of the decal.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

bTransp
If this parameter is TRUE, the decal has a transparent color. If it is FALSE, it has an opaque color. The default value is FALSE.

IDirect3DRMTexture::SetDecalTransparentColor

```
HRESULT SetDecalTransparentColor(D3DCOLOR rcTransp);
```

Sets the transparent color for a decal.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rcTransp
New transparent color. The default transparent color is black.

IDirect3DRMTexture::SetShades

```
HRESULT SetShades(DWORD ulShades);
```

Sets the maximum number of shades to use for each color for the texture when rendering. This method is required only in the ramp color model.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

ulShades
New number of shades. The default value is 16.

IDirect3DRMUserVisual Interface Method Groups

Applications use the **IDirect3DRMUserVisual** interface to initialize Direct3DRMUserVisual objects. Note that this initialization is not necessary if the application calls the IDirect3DRM::CreateUserVisual method; it is required only if the application calls the IDirect3DRM::CreateObject method to create the user-visual object. This interface supports the following methods:

Initialization	Init
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMUserVisual object without affecting the functionality of the original interface. In addition, the **IDirect3DRMUserVisual** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMUserVisual object is obtained by calling the IDirect3DRM::CreateUserVisual method.

IDirect3DRMUserVisual::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMUserVisual object by 1. This method is part of the IUnknown interface inherited by Direct3DRMUserVisual.

- Returns the new reference count of the object.

When the Direct3DRMUserVisual object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMUserVisual::Release method to decrease the reference count of the object by 1.

IDirect3DRMUserVisual::Init

```
HRESULT Init(D3DRMUSERVISUALCALLBACK d3drmUVProc, void * lpArg);
```

Initializes a Direct3DRMUserVisual object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmUVProc

Application-defined D3DRMUSERVISUALCALLBACK callback function.

lpArg

Application-defined data to be passed to the callback function.

Applications can call the IDirect3DRM::CreateUserVisual method to create and initialize a user-visual object at the same time. It is necessary to call **IDirect3DRMUserVisual::Init** only when the application has created the user-visual object by calling the IDirect3DRM::CreateObject method.

IDirect3DRMUserVisual::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMUserVisual object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMUserVisual.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMUserVisual::QueryInterface** method allows Direct3DRMUserVisual objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMUserVisual::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMUserVisual object by 1. This method is part of the IUnknown interface inherited by Direct3DRMUserVisual.

- Returns the new reference count of the object.

The Direct3DRMUserVisual object deallocates itself when its reference count reaches 0. Use the IDirect3DRMUserVisual::AddRef method to increase the reference count of the object by 1.

IDirect3DRMViewport Interface Method Groups

Applications use the methods of the **IDirect3DRMViewport** interface to work with viewport objects. The methods can be organized into the following groups:

Camera	GetCamera SetCamera
Clipping planes	GetBack GetFront GetPlane SetBack SetFront SetPlane
Dimensions	GetHeight GetWidth
Field of view	GetField SetField
Initialization	Init
IUnknown	AddRef QueryInterface Release
Miscellaneous	Clear Configure ForceUpdate GetDevice GetDirect3DViewport Pick Render
Offsets	GetX GetY
Projection types	GetProjection SetProjection
Scaling	GetUniformScaling SetUniformScaling
Transformations	InverseTransform Transform

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMViewport object without affecting the functionality of the original interface. In addition, the **IDirect3DRMViewport** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMViewport object is obtained by calling the IDirect3DRM::CreateViewport method.

IDirect3DRMViewport::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMViewport object by 1. This method is part of the IUnknown interface inherited by Direct3DRMViewport.

- Returns the new reference count of the object.

When the Direct3DRMViewport object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMViewport::Release method to decrease the reference count of the object by 1.

IDirect3DRMViewport::Clear

```
HRESULT Clear();
```

Clears the given viewport to the current background color.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::Configure

```
HRESULT Configure(LONG lX, LONG lY, DWORD dwWidth, DWORD dwHeight);
```

Reconfigures the origin and dimensions of a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lX and *lY*

New position of the viewport.

dwWidth and *dwHeight*

New width and height of the viewport.

This method returns D3DRMERR_BADVALUE if *lX* + *dwWidth* or *lY* + *dwHeight* are greater than the width or height of the device, or if any of *lX*, *lY*, *dwWidth*, or *dwHeight* is less than zero.

IDirect3DRMViewport::ForceUpdate

```
HRESULT ForceUpdate(DWORD dwX1, DWORD dwY1, DWORD dwX2,
    DWORD dwY2);
```

Forces an area of the viewport to be updated. The specified area will be copied to the screen at the next call to the IDirect3DRMDevice::Update method.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

dwX1 and *dwY1*

Upper-left corner of area to be updated.

dwX2 and *dwY2*

Lower-right corner of area to be updated.

The system might update any region that is larger than the specified rectangle, including possibly the entire window.

IDirect3DRMViewport::GetBack

```
D3DVALUE GetBack();
```

Retrieves the position of the back clipping plane for a viewport.

- Returns a value describing the position.

IDirect3DRMViewport::GetCamera

```
HRESULT GetCamera(LPDIRECT3DRMFRAME *lpCamera);
```

Retrieves the camera for a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpCamera
Address of a variable that represents the Direct3DRMFrame object representing the camera.

IDirect3DRMViewport::GetDevice

```
HRESULT GetDevice(LPDIRECT3DRMDEVICE *lpD3DRMDevice);
```

Retrieves the device associated with a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMDevice
Address of a variable that represents the Direct3DRMDevice object.

IDirect3DRMViewport::GetDirect3DViewport

```
HRESULT GetDirect3DViewport(LPDIRECT3DVIEWPORT * lplpD3DViewport);
```

Retrieves the Direct3D viewport corresponding to the current Direct3DRMViewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lplpD3DViewport
Address of a pointer that is initialized with a pointer to the Direct3DViewport object.

IDirect3DRMViewport::GetField

```
D3DVALUE GetField();
```

Retrieves the field of view for a viewport.

- Returns a value describing the field of view.

IDirect3DRMViewport::GetFront

```
D3DVALUE GetFront();
```

Retrieves the position of the front clipping plane for a viewport.

- Returns a value describing the position.

IDirect3DRMViewport::GetHeight

```
DWORD GetHeight();
```

Retrieves the height, in pixels, of the viewport.

- Returns the pixel height.

IDirect3DRMViewport::GetPlane

```
HRESULT GetPlane(D3DVALUE *lpd3dvLeft, D3DVALUE *lpd3dvRight,
    D3DVALUE *lpd3dvBottom, D3DVALUE *lpd3dvTop);
```

Retrieves the front clipping plane of the viewing frustum.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpd3dvLeft, lpd3dvRight, lpd3dvBottom, and *lpd3dvTop*
Addresses of variables that will be filled to represent the front clipping plane.

IDirect3DRMViewport::GetProjection

```
D3DRMPROJECTIONTYPE GetProjection();
```

Retrieves the projection type for the viewport. A viewport can use either orthographic or perspective projection.

- Returns one of the members of the D3DRMPROJECTIONTYPE enumerated type.

IDirect3DRMViewport::GetUniformScaling

```
BOOL GetUniformScaling();
```

Retrieves the scaling property used to scale the viewing volume into the larger dimension of the window.

- Returns TRUE if the viewport scales uniformly, or FALSE otherwise.

IDirect3DRMViewport::GetWidth

```
DWORD GetWidth();
```

Retrieves the width, in pixels, of the viewport.

- Returns the pixel width.

IDirect3DRMViewport::GetX

```
LONG GetX();
```

Retrieves the x-offset of the start of the viewport on a device.

- Returns the x-offset.

IDirect3DRMViewport::GetY

```
LONG GetY();
```

Retrieves the y-offset of the start of the viewport on a device.

- Returns the y-offset.

IDirect3DRMViewport::Init

```
HRESULT Init(LPDIRECT3DRMDEVICE lpD3DRMDevice,
    LPDIRECT3DRMFRAME lpD3DRMFrameCamera, DWORD xpos, DWORD ypos,
    DWORD width, DWORD height);
```

Initializes a Direct3DRMViewport object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMDevice
Address of the DirectD3DRMDevice object associated with this viewport.

lpD3DRMFrameCamera
Address of the camera frame associated with this viewport.

xpos and *ypos*
The x- and y-coordinates of the upper-left corner of the viewport.

width and *height*
Width and height of the viewport.

IDirect3DRMViewport::InverseTransform

```
HRESULT InverseTransform(D3DVECTOR * lprvDst, D3DRMVECTOR4D * lprvSrc);
```

Transforms the vector in the *lprvSrc* parameter in screen coordinates to world coordinates, and returns the result in the *lprvDst* parameter.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lprvDst
Address of a D3DVECTOR structure that will be filled with the result of the operation when the method returns.

lprvSrc
Address of a D3DRMVECTOR4D structure representing the source of the operation.

IDirect3DRMViewport::Pick

```
HRESULT Pick(LONG lX, LONG lY,
    LPDIRECT3DRMPICKEDARRAY* lplpVisuals);
```

Finds a depth-sorted list of objects (and faces, if relevant) that includes the path taken in the hierarchy from the root down to the frame that contained the object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lX and *lY*

Coordinates to be used for picking.

lplpVisuals

Address of a pointer to be initialized with a valid pointer to the IDirect3DRMPickedArray interface if the call succeeds.

IDirect3DRMViewport::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMViewport object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMViewport.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMViewport::QueryInterface** method allows Direct3DRMViewport objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMViewport::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMViewport object by 1. This method is part of the IUnknown interface inherited by Direct3DRMViewport.

- Returns the new reference count of the object.

The Direct3DRMViewport object deallocates itself when its reference count reaches 0. Use the IDirect3DRMViewport::AddRef method to increase the reference count of the object by 1.

IDirect3DRMViewport::Render

```
HRESULT Render(LPDIRECT3DRMFRAME lpD3DRMFrame);
```

Renders a frame hierarchy to the given viewport. Only those visuals on the given frame and any frames below it in the hierarchy are rendered.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpD3DRMFrame

Address of a variable that represents the Direct3DRMFrame object that represents the frame hierarchy to be rendered.

IDirect3DRMViewport::SetBack

```
HRESULT SetBack(D3DVALUE rvBack);
```

Sets the position of the back clipping plane for a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvBack
New position of the back clipping plane.

IDirect3DRMViewport::SetCamera

```
HRESULT SetCamera(LPDIRECT3DRMFRAME lpCamera);
```

Sets a camera for a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpCamera

Address of a variable that represents the Direct3DRMFrame object that represents the camera.

This method sets a viewport's position, direction, and orientation to that of the given camera frame. The view is oriented along the positive z-axis of the camera frame, with the up direction being in the direction of the positive y-axis.

IDirect3DRMViewport::SetField

```
HRESULT SetField(D3DVALUE rvField);
```

Sets the field of view for a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvField

New field of view. The default value is 0.5. If this value is less than or equal to zero, this method returns the D3DRMERR_BADVALUE error.

IDirect3DRMViewport::SetFront

```
HRESULT SetFront(D3DVALUE rvFront);
```

Sets the position of the front clipping plane for a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvFront

New position of the front clipping plane.

The default position is 1.0. If the value passed is less than or equal to zero, this method returns the D3DRMERR_BADVALUE error.

IDirect3DRMViewport::SetPlane

```
HRESULT SetPlane(D3DVALUE rvLeft, D3DVALUE rvRight, D3DVALUE rvBottom,
    D3DVALUE rvTop);
```

Sets the front clipping plane of the viewing frustum by supplying the coordinates of the four sides, relative to the camera's z-axis.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rvLeft, rvRight, rvBottom, and *rvTop*
New front clipping plane.

IDirect3DRMViewport::SetProjection

```
HRESULT SetProjection(D3DRMPROJECTIONTYPE rptType);
```

Sets the projection type for a viewport.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

rptType
One of the members of the D3DRMPROJECTIONTYPE enumerated type.

IDirect3DRMViewport::SetUniformScaling

```
HRESULT SetUniformScaling(BOOL bScale);
```

Sets the scaling property used to scale the viewing volume into the larger dimension of the window.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

bScale

New scaling property. If this parameter is TRUE, the same horizontal and vertical scaling factor is used to scale the viewing volume. Otherwise, different scaling factors are used to scale the viewing volume exactly into the window. The default setting is TRUE.

This method is typically used with the IDirect3DRMViewport::SetPlane method to support banding.

IDirect3DRMViewport::Transform

```
HRESULT Transform(D3DRMVECTOR4D * lprvDst, D3DVECTOR * lprvSrc);
```

Transforms the vector in the *lprvSrc* parameter in world coordinates to screen coordinates, and returns the result in the *lprvDst* parameter.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lprvDst
Address of a D3DRMVECTOR4D structure that acts as the destination for the transformation operation.

lprvSrc
Address of a D3DVECTOR structure that acts as the source for the transformation operation.

The result of the transformation is a four-element homogeneous vector to avoid dividing by zero when the vector is close to the camera's position. The point represented by the resulting vector is visible if the following equations are true:

$$wx_{min} \le x < wx_{max}$$
$$wy_{min} \le y < wy_{max}$$
$$0 \le z < w$$

where

$$x_{min} = viewport_x - viewport_{width} / 2$$
$$x_{max} = viewport_x + viewport_{width} / 2$$
$$y_{min} = viewport_y - viewport_{height} / 2$$
$$y_{max} = viewport_y + viewport_{height} / 2$$

IDirect3DRMWinDevice Interface Method Groups

Applications use the methods of the **IDirect3DRMWinDevice** interface to respond to window messages in a window procedure. This interface supports the following methods:

Handles	HandleActivate
	HandlePaint
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMWinDevice object without affecting the functionality of the original interface.

The Direct3DRMWinDevice object is obtained by calling the IDirect3DRMObject::QueryInterface method and specifying IID_IDirect3DRMWinDevice, or by calling a method such as IDirect3DRM::CreateDeviceFromD3D. Its methods are inherited from the IDirect3DRMDevice interface.

IDirect3DRMWinDevice::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMWinDevice object by 1. This method is part of the IUnknown interface inherited by Direct3DRMWinDevice.

- Returns the new reference count of the object.

When the Direct3DRMWinDevice object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMWinDevice::Release method to decrease the reference count of the object by 1.

IDirect3DRMWinDevice::HandleActivate

```
HRESULT HandleActivate(WORD wParam);
```

Responds to a Windows WM_ACTIVATE message. This ensures that the colors are correct in the active rendering window.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

wParam

WPARAM parameter passed to the message-processing procedure with the WM_ACTIVATE message.

IDirect3DRMWinDevice::HandlePaint

```
HRESULT HandlePaint(HDC hDC);
```

Responds to a Windows WM_PAINT message. The *hDC* parameter should be taken from the **PAINTSTRUCT** structure given to the Windows **BeginPaint** function. This method should be called before repainting any application areas in the window because it may repaint areas outside the viewports that have been created on the device.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

hDC
Handle of the device context (DC).

IDirect3DRMWinDevice::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMWinDevice object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMWinDevice.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMWinDevice::QueryInterface** method allows Direct3DRMWinDevice objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMWinDevice::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMWinDevice object by 1. This method is part of the IUnknown interface inherited by Direct3DRMWinDevice.

- Returns the new reference count of the object.

The Direct3DRMWinDevice object deallocates itself when its reference count reaches 0. Use the IDirect3DRMWinDevice::AddRef method to increase the reference count of the object by 1.

IDirect3DRMWrap Interface Method Groups

Applications use the methods of the **IDirect3DRMWrap** interface to work with wrap objects. This interface supports the following methods:

Initialization	Init
IUnknown	AddRef QueryInterface Release
Wrap	Apply ApplyRelative

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DRMWrap object without affecting the functionality of the original interface. In addition, the **IDirect3DRMWrap** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMWrap object is obtained by calling the IDirect3DRM::CreateWrap method.

IDirect3DRMWrap::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DRMWrap object by 1. This method is part of the IUnknown interface inherited by Direct3DRMWrap.

- Returns the new reference count of the object.

When the Direct3DRMWrap object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DRMWrap::Release method to decrease the reference count of the object by 1.

IDirect3DRMWrap::Apply

```
HRESULT Apply(LPDIRECT3DRMOBJECT lpObject);
```

Applies a Direct3DRMWrap object to its destination object. The destination object is typically a face or a mesh.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

lpObject
Address of the destination object.

IDirect3DRMWrap::ApplyRelative

```
HRESULT ApplyRelative(LPDIRECT3DRMFRAME frame,
    LPDIRECT3DRMOBJECT mesh);
```

Applies the wrap to the vertices of the object, first transforming each vertex by the frame's world transformation and the wrap's reference frame's inverse world transformation.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

frame

Direct3DRMFrame object containing the object to wrap.

mesh

Direct3DRMWrap object to apply.

IDirect3DRMWrap::Init

```
HRESULT Init(D3DRMWRAPTYPE d3drmwt, LPDIRECT3DRMFRAME lpd3drmfRef,
    D3DVALUE ox, D3DVALUE oy, D3DVALUE oz,
    D3DVALUE dx, D3DVALUE dy, D3DVALUE dz,
    D3DVALUE ux, D3DVALUE uy, D3DVALUE uz,
    D3DVALUE ou, D3DVALUE ov, D3DVALUE su, D3DVALUE sv);
```

Initializes a Direct3DRMWrap object.

- Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

d3drmwt
One of the members of the D3DRMWRAPTYPE enumerated type.

lpd3drmfRef
Address of a Direct3DRMFrame object representing the reference frame for this Direct3DRMWrap object.

ox, *oy*, and *oz*
Origin of the wrap.

dx, *dy*, and *dz*
The z-axis of the wrap.

ux, *uy*, and *uz*
The y-axis of the wrap.

ou and *ov*
Origin in the texture.

su and *sv*
Scale factor in the texture.

IDirect3DRMWrap::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DRMWrap object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DRMWrap.

- Returns D3DRM_OK if successful, or D3DRMERR_BADVALUE otherwise.

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DRMWrap::QueryInterface** method allows Direct3DRMWrap objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DRMWrap::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DRMWrap object by 1. This method is part of the IUnknown interface inherited by Direct3DRMWrap.

- Returns the new reference count of the object.

The Direct3DRMWrap object deallocates itself when its reference count reaches 0. Use the IDirect3DRMWrap::AddRef method to increase the reference count of the object by 1.

D3DRMBOX

```
typedef struct _D3DRMBOX {
    D3DVECTOR min, max;
}D3DRMBOX;
typedef D3DRMBOX *LPD3DRMBOX;
```

Defines the bounding box retrieved by the IDirect3DRMMesh::GetBox and IDirect3DRMMeshBuilder::GetBox methods.

min and **max**
Values defining the bounds of the box. These values are D3DVECTOR structures.

D3DRMIMAGE

```
typedef struct _D3DRMIMAGE {
    int                 width, height;
    int                 aspectx, aspecty;
    int                 depth;
    int                 rgb;
    int                 bytes_per_line;
    void*               buffer1;
    void*               buffer2;
    unsigned long       red_mask;
    unsigned long       green_mask;
    unsigned long       blue_mask;
    unsigned long       alpha_mask;
    int                 palette_size;
    D3DRMPALETTEENTRY* palette;
}D3DRMIMAGE;
typedef D3DRMIMAGE, *LPD3DRMIMAGE;
```

Describes an image that is attached to a texture by the IDirect3DRM::CreateTexture method. IDirect3DRMTexture::GetImage returns the address of this image.

width and **height**
Width and height of the image, in pixels.

aspectx and **aspecty**
Aspect ratio for nonsquare pixels.

depth
Bits per pixel.

rgb
If this member FALSE, pixels are indices into a palette. Otherwise, pixels encode RGB values.

bytes_per_line
Number of bytes of memory for a scanline. This value must be a multiple of four.

buffer1
Memory to render into (first buffer).

buffer2
Second rendering buffer for double buffering. Set this member to NULL for single buffering.

red_mask, **green_mask**, **blue_mask**, and **alpha_mask**
If **rgb** is TRUE, these members are masks for the red, green, and blue parts of a pixel. Otherwise, they are masks for the significant bits of the red, green, and blue elements in the palette. For example, most SVGA displays use 64 intensities of red, green, and blue, so the masks should all be set to 0xfc.

palette_size
Number of entries in the palette.

palette
If **rgb** is FALSE, this member is the address of a D3DRMPALETTEENTRY structure describing the palette entry.

D3DRMLOADMEMORY

```
typedef struct _D3DRMLOADMEMORY {
    LPVOID lpMemory;
    DWORD  dSize;
} D3DRMLOADMEMORY, *LPD3DRMLOADMEMORY;
```

Identifies a resource to be loaded when an application calls the IDirect3DRM::Load method (or one of the other **Load** methods) and specifies D3DRMLOAD_FROMMEMORY.

lpMemory
Address of a block of memory to be loaded.

dSize
Size, in bytes, of the block of memory to be loaded.

D3DRMLOADRESOURCE

```
typedef struct _D3DRMLOADRESOURCE {
    HMODULE hModule;
    LPCTSTR lpName;
    LPCTSTR lpType;
} D3DRMLOADRESOURCE, *LPD3DRMLOADRESOURCE;
```

Identifies a resource to be loaded when an application calls the IDirect3DRM::Load method (or one of the other **Load** methods) and specifies D3DRMLOAD_FROMRESOURCE.

hModule
Handle of the module containing the resource to be loaded. If this member is NULL, the resource must be attached to the calling executable file.

lpName
Name of the resource to be loaded. For example, if the resource is a mesh, this member should specify the name of the mesh file.

lpType
User-defined type identifying the resource.

If the high-order word of the **lpName** or **lpType** member is zero, the low-order word specifies the integer identifier of the name or type of the given resource. Otherwise, those parameters are long pointers to null-terminated strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer identifier of the resource's name or type. For example, the string "#258" represents the integer identifier 258. An application should reduce the amount of memory required for the resources by referring to them by integer identifier instead of by name.

When an application calls a **Load** method and specifies D3DRMLOAD_FROMRESOURCE, it does not need to find or unlock any resources; the system handles this automatically.

D3DRMPALETTEENTRY

```
typedef struct _D3DRMPALETTEENTRY {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
    unsigned char flags;
}D3DRMPALETTEENTRY;
typedef D3DRMPALETTEENTRY, *LPD3DRMPALETTEENTRY;
```

Describes the color palette used in a D3DRMIMAGE structure. This structure is used only if the **rgb** member of the **D3DRMIMAGE** structure is FALSE. (If it is TRUE, RGB values are used.)

red, **green**, and **blue**

Values defining the primary color components that define the palette. These values can range from 0 through 255.

flags

Value defining how the palette is used by the renderer. This value is one of the members of the D3DRMPALETTEFLAGS enumerated type.

D3DRMPICKDESC

```
typedef struct _D3DRMPICKDESC {
    ULONG       ulFaceIdx;
    LONG        lGroupIdx;
    D3DVECTOR   vPosition;
} D3DRMPICKDESC, *LPD3DRMPICKDESC;
```

Contains the pick position and face and group identifiers of the objects retrieved by the IDirect3DRMPickedArray::GetPick method.

ulFaceIdx
Face index of the retrieved object.

lGroupIdx
Group index of the retrieved object.

vPosition
Value describing the position of the retrieved object. This value is a D3DVECTOR structure.

D3DRMQUATERNION

```
typedef struct _D3DRMQUATERNION {
    D3DVALUE   s;
    D3DVECTOR  v;
}D3DRMQUATERNION;
typedef D3DRMQUATERNION, *LPD3DRMQUATERNION;
```

Describes the rotation used by the IDirect3DRMAnimation::AddRotateKey method. It is also used in several of Direct3D's mathematical functions.

D3DRMVECTOR4D

```
typedef struct _D3DRMVECTOR4D {
    D3DVALUE x;
    D3DVALUE y;
    D3DVALUE z;
    D3DVALUE w;
}D3DRMVECTOR4D;
typedef D3DRMVECTOR4D, *LPD3DRMVECTOR4D;
```

Describes the screen coordinates used as the destination of a transformation by the IDirect3DRMViewport::Transform method and as the source of a transformation by the IDirect3DRMViewport::InverseTransform method.

x, **y**, **z**, and **w**
Values of the D3DVALUE type describing homogeneous values. These values define the result of the transformation.

D3DRMVERTEX

```
typedef struct _D3DRMVERTEX{
    D3DVECTOR position;
    D3DVECTOR normal;
    D3DVALUE  tu, tv;
    D3DCOLOR  color;
} D3DRMVERTEX;
```

Describes a vertex in a Direct3DRMMesh object.

position
Position of the vertex.

normal
Normal vector for the vertex.

tu and **tv**
Horizontal and vertical texture coordinates, respectively, for the vertex.

color
Vertex color.

D3DRMCOLORSOURCE

```
typedef enum _D3DRMCOLORSOURCE{
    D3DRMCOLOR_FROMFACE,
    D3DRMCOLOR_FROMVERTEX
} D3DRMCOLORSOURCE;
```

Describes the color source of a Direct3DRMMeshBuilder object. You can set the color source by using the IDirect3DRMMeshBuilder::SetColorSource method. To retrieve it, use the IDirect3DRMMeshBuilder::GetColorSource method.

D3DRMCOLOR_FROMFACE
The object's color source is a face.

D3DRMCOLOR_FROMVERTEX
The object's color source is a vertex.

D3DRMCOMBINETYPE

```
typedef enum _D3DRMCOMBINETYPE{
    D3DRMCOMBINE_REPLACE,
    D3DRMCOMBINE_BEFORE,
    D3DRMCOMBINE_AFTER
} D3DRMCOMBINETYPE;
```

Specifies how to combine two matrices.

D3DRMCOMBINE_REPLACE
The supplied matrix replaces the frame's current matrix.

D3DRMCOMBINE_BEFORE
The supplied matrix is multiplied with the frame's current matrix and precedes the current matrix in the calculation.

D3DRMCOMBINE_AFTER
The supplied matrix is multiplied with the frame's current matrix and follows the current matrix in the calculation.

The order of the supplied and current matrices when they are multiplied together is important because matrix multiplication is not commutative.

D3DRMFILLMODE

```
typedef enum _D3DRMFILLMODE {
    D3DRMFILL_POINTS    = 0 * D3DRMLIGHT_MAX,
    D3DRMFILL_WIREFRAME = 1 * D3DRMLIGHT_MAX,
    D3DRMFILL_SOLID     = 2 * D3DRMLIGHT_MAX,
    D3DRMFILL_MASK      = 7 * D3DRMLIGHT_MAX,
    D3DRMFILL_MAX       = 8 * D3DRMLIGHT_MAX
} D3DRMFILLMODE;
```

One of the enumerated types that is used in the definition of the D3DRMRENDERQUALITY type.

D3DRMFILL_POINTS
Fills points only; minimum fill mode.

D3DRMFILL_WIREFRAME
Fill wireframes.

D3DRMFILL_SOLID
Fill solid objects.

D3DRMFILL_MASK
Fill using a mask.

D3DRMFILL_MAX
Maximum value for fill mode.

D3DRMFOGMODE

```
typedef enum _D3DRMFOGMODE{
    D3DRMFOG_LINEAR,
    D3DRMFOG_EXPONENTIAL,
    D3DRMFOG_EXPONENTIALSQUARED
} D3DRMFOGMODE;
```

Contains values that specify how rapidly and in what ways the fog effect intensifies with increasing distance from the camera.

D3DRMFOG_LINEAR

The fog effect intensifies linearly between the start and end points, according to the following formula:

$$f = \frac{end - z}{end - start}$$

This is the only fog mode supported for DirectX 2.

D3DRMFOG_EXPONENTIAL

The fog effect intensifies exponentially, according to the following formula:

$$f = e^{-(density \times z)}$$

D3DRMFOG_EXPONENTIALSQUARED

The fog effect intensifies exponentially with the square of the distance, according to the following formula:

$$f = e^{-(density \times z)}$$

Note that fog can be considered a measure of visibility—the lower the fog value produced by one of the fog equations, the less visible an object is.

You can specify the fog's density and start and end points by using the IDirect3DRMFrame::SetSceneFogParams method. In the formulas for the exponential fog modes, *e* is the base of the natural logarithms; its value is approximately 2.71828.

D3DRMFRAMECONSTRAINT

```
typedef enum _D3DRMFRAMECONSTRAINT {
    D3DRMCONSTRAIN_Z,
    D3DRMCONSTRAIN_Y,
    D3DRMCONSTRAIN_X
} D3DRMFRAMECONSTRAINT;
```

Describes the axes of rotation to constrain when viewing a Direct3DRMFrame object. The IDirect3DRMFrame::LookAt method uses this enumerated type.

D3DRMCONSTRAIN_Z
Use only x and y rotations.

D3DRMCONSTRAIN_Y
Use only x and z rotations.

D3DRMCONSTRAIN_X
Use only y and z rotations.

D3DRMLIGHTMODE

```
typedef enum _D3DRMLIGHTMODE {
    D3DRMLIGHT_OFF      = 0 * D3DRMSHADE_MAX,
    D3DRMLIGHT_ON       = 1 * D3DRMSHADE_MAX,
    D3DRMLIGHT_MASK     = 7 * D3DRMSHADE_MAX,
    D3DRMLIGHT_MAX      = 8 * D3DRMSHADE_MAX
} D3DRMLIGHTMODE;
```

One of the enumerated types that is used in the definition of the D3DRMRENDERQUALITY type.

D3DRMLIGHT_OFF
Lighting is off.

D3DRMLIGHT_ON
Lighting is on.

D3DRMLIGHT_MASK
Lighting uses a mask.

D3DRMLIGHT_MAX
Maximum lighting mode.

D3DRMLIGHTTYPE

```
typedef enum _D3DRMLIGHTTYPE{
    D3DRMLIGHT_AMBIENT,
    D3DRMLIGHT_POINT,
    D3DRMLIGHT_SPOT,
    D3DRMLIGHT_DIRECTIONAL,
    D3DRMLIGHT_PARALLELPOINT
} D3DRMLIGHTTYPE;
```

Defines the light type in calls to the IDirect3DRM::CreateLight method.

D3DRMLIGHT_AMBIENT
Light is ambient.

D3DRMLIGHT_POINT
Light is a point source.

D3DRMLIGHT_SPOT
Light is a spotlight source.

D3DRMLIGHT_DIRECTIONAL
Light is a directional source.

D3DRMLIGHT_PARALLELPOINT
Light is a parallel source.

D3DRMMATERIALMODE

```
typedef enum _D3DRMMATERIALMODE{
    D3DRMMATERIAL_FROMMESH,
    D3DRMMATERIAL_FROMPARENT,
    D3DRMMATERIAL_FROMFRAME
} D3DRMMATERIALMODE;
```

Describes the type retrieved by the IDirect3DRMFrame::GetMaterialMode method and set by the IDirect3DRMFrame::SetMaterialMode method.

D3DRMMATERIAL_FROMMESH
Material information is retrieved from the visual object (the mesh) itself. This is the default setting.

D3DRMMATERIAL_FROMPARENT
Material information, along with color or texture information, is inherited from the parent frame.

D3DRMMATERIAL_FROMFRAME
Material information is retrieved from the frame, overriding any previous material information that the visual object may have possessed.

D3DRMPALETTEFLAGS

```
typedef enum _D3DRMPALETTEFLAGS {
    D3DRMPALETTE_FREE,
    D3DRMPALETTE_READONLY,
    D3DRMPALETTE_RESERVED
} D3DRMPALETTEFLAGS;
```

Used to define how a color may be used in the D3DRMPALETTEENTRY structure.

D3DRMPALETTE_FREE
Renderer may use this entry freely.

D3DRMPALETTE_READONLY
Fixed but may be used by renderer.

D3DRMPALETTE_RESERVED
May not be used by renderer.

D3DRMPROJECTIONTYPE

```
typedef enum _D3DRMPROJECTIONTYPE{
    D3DRMPROJECT_PERSPECTIVE,
    D3DRMPROJECT_ORTHOGRAPHIC
} D3DRMPROJECTIONTYPE;
```

Defines the type of projection used in a Direct3DRMViewport object. The IDirect3DRMViewport::GetProjection and IDirect3DRMViewport::SetProjection methods use this enumerated type.

D3DRMPROJECT_PERSPECTIVE
The projection is perspective.

D3DRMPROJECT_ORTHOGRAPHIC
The projection is orthographic.

D3DRMRENDERQUALITY

```
typedef enum _D3DRMSHADEMODE {
    D3DRMSHADE_FLAT        = 0,
    D3DRMSHADE_GOURAUD     = 1,
    D3DRMSHADE_PHONG       = 2,
    D3DRMSHADE_MASK        = 7,
    D3DRMSHADE_MAX         = 8
} D3DRMSHADEMODE;
typedef enum _D3DRMLIGHTMODE {
    D3DRMLIGHT_OFF          = 0 * D3DRMSHADE_MAX,
    D3DRMLIGHT_ON           = 1 * D3DRMSHADE_MAX,
    D3DRMLIGHT_MASK         = 7 * D3DRMSHADE_MAX,
    D3DRMLIGHT_MAX          = 8 * D3DRMSHADE_MAX
} D3DRMLIGHTMODE;
typedef enum _D3DRMFILLMODE {
    D3DRMFILL_POINTS        = 0 * D3DRMLIGHT_MAX,
    D3DRMFILL_WIREFRAME     = 1 * D3DRMLIGHT_MAX,
    D3DRMFILL_SOLID         = 2 * D3DRMLIGHT_MAX,
    D3DRMFILL_MASK          = 7 * D3DRMLIGHT_MAX,
    D3DRMFILL_MAX           = 8 * D3DRMLIGHT_MAX
} D3DRMFILLMODE;

typedef DWORD D3DRMRENDERQUALITY;

#define D3DRMRENDER_WIREFRAME
(D3DRMSHADE_FLAT+D3DRMLIGHT_OFF+D3DRMFILL_WIREFRAME)
#define D3DRMRENDER_UNLITFLAT
(D3DRMSHADE_FLAT+D3DRMLIGHT_OFF+D3DRMFILL_SOLID)
#define D3DRMRENDER_FLAT
(D3DRMSHADE_FLAT+D3DRMLIGHT_ON+D3DRMFILL_SOLID)
#define D3DRMRENDER_GOURAUD
(D3DRMSHADE_GOURAUD+D3DRMLIGHT_ON+D3DRMFILL_SOLID)
#define D3DRMRENDER_PHONG
(D3DRMSHADE_PHONG+D3DRMLIGHT_ON+D3DRMFILL_SOLID)
```

Combines descriptions of the shading mode, lighting mode, and filling mode for a Direct3DRMMesh object.

D3DRMSHADEMODE, D3DRMLIGHTMODE, and D3DRMFILLMODE

These enumerated types describe the shade, light, and fill modes for Direct3DRMMesh objects.

D3DRMRENDER_WIREFRAME

Display only the edges.

D3DRMRENDER_UNLITFLAT

Flat shaded without lighting.

D3DRMRENDER_FLAT

Flat shaded.

D3DRMRENDER_GOURAUD

Gouraud shaded.

D3DRMRENDER_PHONG

Phong shaded. Phong shading is not supported for DirectX 2.

D3DRMSHADEMODE

```
typedef enum _D3DRMSHADEMODE {
    D3DRMSHADE_FLAT     = 0,
    D3DRMSHADE_GOURAUD  = 1,
    D3DRMSHADE_PHONG    = 2,
    D3DRMSHADE_MASK     = 7,
    D3DRMSHADE_MAX      = 8
} D3DRMSHADEMODE;
```

One of the enumerated types that is used in the definition of the D3DRMRENDERQUALITY type.

D3DRMSORTMODE

```
typedef enum _D3DRMSORTMODE {
    D3DRMSORT_FROMPARENT,
    D3DRMSORT_NONE,
    D3DRMSORT_FRONTTOBACK,
    D3DRMSORT_BACKTOFRONT
} D3DRMSORTMODE;
```

Describes how child frames are sorted in a scene.

D3DRMSORT_FROMPARENT
Child frames inherit the sorting order of their parents. This is the default setting.

D3DRMSORT_NONE
Child frames are not sorted.

D3DRMSORT_FRONTTOBACK
Child frames are sorted front-to-back.

D3DRMSORT_BACKTOFRONT
Child frames are sorted back-to-front.

D3DRMTEXTUREQUALITY

```
typedef enum _D3DRMTEXTUREQUALITY{
    D3DRMTEXTURE_NEAREST,
    D3DRMTEXTURE_LINEAR,
    D3DRMTEXTURE_MIPNEAREST,
    D3DRMTEXTURE_MIPLINEAR,
    D3DRMTEXTURE_LINEARMIPNEAREST,
    D3DRMTEXTURE_LINEARMIPLINEAR
} D3DRMTEXTUREQUALITY;
```

Describes the texture quality for the IDirect3DRMDevice::SetTextureQuality and IDirect3DRMDevice::GetTextureQuality methods.

D3DRMTEXTURE_NEAREST
Choose the nearest pixel in the texture.

D3DRMTEXTURE_LINEAR
Linearly interpolate the four nearest pixels.

D3DRMTEXTURE_MIPNEAREST
Similar to D3DRMTEXTURE_NEAREST, but uses the appropriate mipmap instead of the texture.

D3DRMTEXTURE_MIPLINEAR
Similar to D3DRMTEXTURE_LINEAR, but uses the appropriate mipmap instead of the texture.

D3DRMTEXTURE_LINEARMIPNEAREST
Similar to D3DRMTEXTURE_MIPNEAREST, but interpolates between the two nearest mipmaps.

D3DRMTEXTURE_LINEARMIPLINEAR
Similar to D3DRMTEXTURE_MIPLINEAR, but interpolates between the two nearest mipmaps.

D3DRMUSERVISUALREASON

```
typedef enum _D3DRMUSERVISUALREASON {
    D3DRMUSERVISUAL_CANSEE,
    D3DRMUSERVISUAL_RENDER
} D3DRMUSERVISUALREASON;
```

Defines the reason the system has called the D3DRMUSERVISUALCALLBACK callback function.

D3DRMUSERVISUAL_CANSEE
The callback function should return TRUE if the user-visual object is visible in the viewport.

D3DRMUSERVISUAL_RENDER
The callback function should render the user-visual object.

D3DRMWRAPTYPE

```
typedef enum _D3DRMWRAPTYPE{
    D3DRMWRAP_FLAT,
    D3DRMWRAP_CYLINDER,
    D3DRMWRAP_SPHERE,
    D3DRMWRAP_CHROME
} D3DRMWRAPTYPE;
```

Defines the type of Direct3DRMWrap object created by the IDirect3DRM::CreateWrap method. You can also use this enumerated type to initialize a Direct3DRMWrap object in a call to the IDirect3DRMWrap::Init method.

D3DRMWRAP_FLAT
The wrap is flat.

D3DRMWRAP_CYLINDER
The wrap is cylindrical.

D3DRMWRAP_SPHERE
The wrap is spherical.

D3DRMWRAP_CHROME
The wrap allocates texture coordinates so that the texture appears to be reflected onto the objects.

D3DRMXOFFORMAT

```
typedef enum _D3DRMXOFFORMAT{
    D3DRMXOF_BINARY,
    D3DRMXOF_COMPRESSED,
    D3DRMXOF_TEXT
} D3DRMXOFFORMAT;
```

Defines the file type used by the IDirect3DRMMeshBuilder::Save method.

D3DRMXOF_BINARY
Not currently supported.

D3DRMXOF_COMPRESSED
Not currently supported.

D3DRMXOF_TEXT
File is in text format. This is the only file type that is currently supported.

D3DRMZBUFFERMODE

```
typedef enum _D3DRMZBUFFERMODE {
    D3DRMZBUFFER_FROMPARENT,
    D3DRMZBUFFER_ENABLE,
    D3DRMZBUFFER_DISABLE
} D3DRMZBUFFERMODE;
```

Describes whether z-buffering is enabled.

D3DRMZBUFFER_FROMPARENT
The frame inherits the z-buffer setting from its parent frame. This is the default setting.

D3DRMZBUFFER_ENABLE
Z-buffering is enabled.

D3DRMZBUFFER_DISABLE
Z-buffering is disabled.

D3DRMANIMATIONOPTIONS

```
typedef DWORD D3DRMANIMATIONOPTIONS;
#define D3DRMANIMATION_CLOSED             0x02L
#define D3DRMANIMATION_LINEARPOSITION     0x04L
#define D3DRMANIMATION_OPEN               0x01L
#define D3DRMANIMATION_POSITION           0x00000020L
#define D3DRMANIMATION_SCALEANDROTATION   0x00000010L
#define D3DRMANIMATION_SPLINEPOSITION     0x08L
```

Specifies values used by the IDirect3DRMAnimation::GetOptions and IDirect3DRMAnimation::SetOptions methods to define how animations are played.

D3DRMANIMATION_CLOSED
The animation plays continually, looping back to the beginning whenever it reaches the end.

D3DRMANIMATION_LINEARPOSITION
The animation's position is set linearly.

D3DRMANIMATION_OPEN
The animation plays once and stops.

D3DRMANIMATION_POSITION
The animation's position matrix should overwrite any transformation matrices that could be set by other methods.

D3DRMANIMATION_SCALEANDROTATION
The animation's scale and rotation matrix should overwrite any transformation matrices that could be set by other methods.

D3DRMANIMATION_SPLINEPOSITION
The animation's position is set using splines.

D3DRMCOLORMODEL

```
typedef D3DCOLORMODEL D3DRMCOLORMODEL;
```

Describes the color model implemented by the device. For more information, see the D3DCOLORMODEL enumerated type.

D3DRMLOADOPTIONS

```
typedef DWORD D3DRMLOADOPTIONS;
#define D3DRMLOAD_FROMFILE              0x00L
#define D3DRMLOAD_FROMRESOURCE          0x01L
#define D3DRMLOAD_FROMMEMORY            0x02L
#define D3DRMLOAD_FROMSTREAM            0x03L
#define D3DRMLOAD_BYNAME                0x10L
#define D3DRMLOAD_BYPOSITION            0x20L
#define D3DRMLOAD_BYGUID                0x30L
#define D3DRMLOAD_FIRST                 0x40L
#define D3DRMLOAD_INSTANCEBYREFERENCE   0x100L
#define D3DRMLOAD_INSTANCEBYCOPYING     0x200L
```

Defines options for the IDirect3DRM::Load, IDirect3DRMAnimationSet::Load, IDirect3DRMFrame::Load, and IDirect3DRMMeshBuilder::Load methods. These options modify how the object is loaded.

Source flags

D3DRMLOAD_FROMFILE
Load from a file. This is the default setting.

D3DRMLOAD_FROMRESOURCE
Load from a resource. If this flag is specified, the *lpvObjSource* parameter of the calling **Load** method must point to a D3DRMLOADRESOURCE structure.

D3DRMLOAD_FROMMEMORY
Load from memory. If this flag is specified, the *lpvObjSource* parameter of the calling **Load** method must point to a D3DRMLOADMEMORY structure.

D3DRMLOAD_FROMSTREAM
Load from a stream.

Identifier flags

D3DRMLOAD_BYNAME
Load using a specified name.

D3DRMLOAD_BYPOSITION
Load using a specified zero-based position (that is, the *n*th object in the file).

D3DRMLOAD_BYGUID
Load using a specified globally unique identifier (GUID).

D3DRMLOAD_FIRST
Load the first top-level object of the given type (for example, a mesh if the application calls IDirect3DRMMeshBuilder::Load). This is the default setting.

Instance flags

D3DRMLOAD_INSTANCEBYREFERENCE
Check whether an object already exists with the same name as specified and, if so, use an instance of that object instead of creating a new one.

D3DRMLOAD_INSTANCEBYCOPYING
Check whether an object already exists with the same name as specified and, if so, copy that object.

Each of the **Load** methods uses an *lpvObjSource* parameter to specify the source of the object and an *lpvObjID* parameter to identify the object. The system interprets the contents of the *lpvObjSource* parameter based on the choice of source flags, and it interprets the contents of the *lpvObjID* parameter based on the choice of identifier flags.

The instance flags do not change the interpretation of any of the parameters. By using the D3DRMLOAD_INSTANCEBYREFERENCE flag, it is possible for an application to load the same file twice without creating any new objects. If an object does not have a name, setting the D3DRMLOAD_INSTANCEBYREFERENCE flag has the same effect as setting the D3DRMLOAD_INSTANCEBYCOPYING flag; the loader creates each unnamed object as a new one, even if some of the objects are identical.

D3DRMMAPPING

```
typedef DWORD D3DRMMAPPING, D3DRMMAPPINGFLAG;
static const D3DRMMAPPINGFLAG D3DRMMAP_WRAPU = 1;
static const D3DRMMAPPINGFLAG D3DRMMAP_WRAPV = 2;
static const D3DRMMAPPINGFLAG D3DRMMAP_PERSPCORRECT = 4;
```

Specifies values used by the IDirect3DRMMesh::GetGroupMapping and IDirect3DRMMesh::SetGroupMapping methods to define how textures are mapped to a group.

D3DRMMAPPINGFLAG
Type equivalent to **D3DRMMAPPING**.

D3DRMMAP_WRAPU
Texture wraps in the u direction.

D3DRMMAP_WRAPV
Texture wraps in the v direction.

D3DRMMAP_PERSPCORRECT
Texture wrapping is perspective-corrected.

The D3DRMMAP_WRAPU and D3DRMMAP_WRAPV flags determine how the rasterizer interprets texture coordinates. The rasterizer always interpolates the shortest distance between texture coordinates; that is, a line. The path taken by this line, and the valid values for the u- and v-coordinates, varies with the use of the wrapping flags. If either or both flags is set, the line can wrap around the texture edge in the u or v direction, as if the texture had a cylindrical or toroidal topology. For more information, see Direct3DRMWrap.

D3DRMMATRIX4D

```
typedef D3DVALUE D3DRMMATRIX4D[4][4];
```

Expresses a transformation as an array. The organization of the matrix entries is D3DRMMATRIX4D[*row*][*column*].

D3DRMSAVEOPTIONS

```
typedef DWORD D3DRMSAVEOPTIONS;
#define D3DRMXOFSAVE_NORMALS 1
#define D3DRMXOFSAVE_TEXTURECOORDINATES 2
#define D3DRMXOFSAVE_MATERIALS 4
#define D3DRMXOFSAVE_TEXTURENAMES 8
#define D3DRMXOFSAVE_ALL 15
```

Defines options for the IDirect3DRMMeshBuilder::Save method.

D3DRMXOFSAVE_NORMALS
Save normal vectors in addition to the basic geometry.

D3DRMXOFSAVE_TEXTURECOORDINATES
Save texture coordinates in addition to the basic geometry.

D3DRMXOFSAVE_MATERIALS
Save materials in addition to the basic geometry.

D3DRMXOFSAVE_TEXTURENAMES
Save texture names in addition to the basic geometry.

D3DRMXOFSAVE_ALL
Save normal vectors, texture coordinates, materials, and texture names in addition to the basic geometry.

Return Values

The methods of the Direct3D Retained-Mode Component Object Model (COM) interfaces can return the following values.

D3DRM_OK
: No error.

D3DRMERR_BADALLOC
: Out of memory.

D3DRMERR_BADDEVICE
: Device is not compatible with renderer.

D3DRMERR_BADFILE
: Data file is corrupt.

D3DRMERR_BADMAJORVERSION
: Bad DLL major version.

D3DRMERR_BADMINORVERSION
: Bad DLL minor version.

D3DRMERR_BADOBJECT
: Object expected in argument.

D3DRMERR_BADTYPE
: Bad argument type passed.

D3DRMERR_BADVALUE
: Bad argument value passed.

D3DRMERR_FACEUSED
: Face already used in a mesh.

D3DRMERR_FILENOTFOUND
: File cannot be opened.

D3DRMERR_NOTDONEYET
: Unimplemented.

D3DRMERR_NOTFOUND
: Object not found in specified place.

D3DRMERR_UNABLETOEXECUTE
: Unable to carry out procedure.

D3DDivide

```
D3DDivide(a, b)    (float)((double) (a) / (double) (b))
```

Divides two values.

- Returns the quotient of the division.

a and *b*
: Dividend and divisor in the expression, respectively.

D3DMultiply

```
D3DMultiply(a, b)     ((a) * (b))
```

Multiplies two values.

- Returns the product of the multiplication.

a and *b*
: Values to be multiplied.

D3DRGB

```
D3DRGB(r, g, b) \
    (0xff000000L | ( ((long)((r) * 255)) << 16) | \
    (((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Initializes a color with the supplied RGB values.

- Returns the D3DCOLOR value corresponding to the supplied RGB values.

r, *g*, and *b*
Red, green, and blue components of the color. These should be floating-point values in the range 0 through 1.

D3DRGBA

```
D3DRGBA(r, g, b, a) \
    ((((long)((a) * 255)) << 24) | (((long)((r) * 255)) << 16) |
    (((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Initializes a color with the supplied RGBA values.

- Returns the D3DCOLOR value corresponding to the supplied RGBA values.

r, *g*, *b*, and *a*
Red, green, blue, and alpha components of the color.

D3DSTATE_OVERRIDE

```
D3DSTATE_OVERRIDE(type) ((DWORD) (type) + D3DSTATE_OVERRIDE_BIAS)
```

Overrides the state of the rasterization, lighting, or transformation module. Applications can use this macro to lock and unlock a state.

- No return value.

type
State to override. This parameter should be one of the members of the D3DTRANSFORMSTATETYPE, D3DLIGHTSTATETYPE, or D3DRENDERSTATETYPE enumerated types.

An application might, for example, use the STATE_DATA macro (defined in the D3dmacs.h header file in the Misc directory of the DirectX 2 SDK sample code) and D3DSTATE_OVERRIDE to lock and unlock the D3DRENDERSTATE_SHADEMODE render state:

```
//  Lock the shade mode.

STATE_DATA(D3DSTATE_OVERRIDE(D3DRENDERSTATE_SHADEMODE), TRUE, lpBuffer);

//  Work with the shade mode and unlock it when read-only status is not
required.

STATE_DATA(D3DSTATE_OVERRIDE(D3DRENDERSTATE_SHADEMODE), FALSE, lpBuffer);
```

For more information about overriding rendering states, see States and State Overrides.

D3DVAL

```
D3DVAL(val)     ((float)val)
```

Creates a value whose type is D3DVALUE.

- Returns the converted value.

val
Value to be converted.

D3DVALP

```
D3DVALP(val, prec)    ((float)val)
```

Creates a value of the specified precision.

- Returns the converted value.

val
Value to be converted.

prec
Ignored.

The precision, as implemented by the D3DVAL macro, is 16 bits for the fractional part of the value.

RGB_GETBLUE

```
RGB_GETBLUE(rgb)    ((rgb) & 0xff)
```

Retrieves the blue component of a D3DCOLOR value.

- Returns the blue component.

rgb
Color index from which the blue component is retrieved.

RGB_GETGREEN

```
RGB_GETGREEN(rgb)     (((rgb) >> 8) & 0xff)
```

Retrieves the green component of a D3DCOLOR value.

- Returns the green component.

rgb
Color index from which the green component is retrieved.

RGB_GETRED

```
RGB_GETRED(rgb)     (((rgb) >> 16) & 0xff)
```

Retrieves the red component of a D3DCOLOR value.

- Returns the red component.

rgb
Color index from which the red component is retrieved.

RGB_MAKE

```
RGB_MAKE(r, g, b)     ((D3DCOLOR) (((r) << 16) | ((g) << 8) | (b)))
```

Creates an RGB color from supplied values.

- Returns the color.

r, *g*, and *b*
: Red, green, and blue components of the color to be created. These should be integer values in the range zero through 255.

RGB_TORGBA

```
RGB_TORGBA(rgb)     ((D3DCOLOR) ((rgb) | 0xff000000))
```

Creates an RGBA color from a supplied RGB color.

- Returns the RGBA color.

rgb
RGB color to be converted to an RGBA color.

RGBA_GETALPHA

```
RGBA_GETALPHA(rgb)     ((rgb) >> 24)
```

Retrieves the alpha component of an RGBA D3DCOLOR value.

- Returns the alpha component.

rgb
Color index from which the alpha component is retrieved.

RGBA_GETBLUE

```
RGB_GETBLUE(rgb)     ((rgb) & 0xff)
```

Retrieves the blue component of an RGBA D3DCOLOR value.

- Returns the blue component.

rgb
Color index from which the blue component is retrieved.

RGBA_GETGREEN

```
RGB_GETGREEN(rgb)     (((rgb) >> 8) & 0xff)
```

Retrieves the green component of an RGBA D3DCOLOR value.

- Returns the green component.

rgb
Color index from which the green component is retrieved.

RGBA_GETRED

```
RGB_GETRED(rgb)     (((rgb) >> 16) & 0xff)
```

Retrieves the red component of an RGBA D3DCOLOR value.

- Returns the red component.

rgb
Color index from which the red component is retrieved.

RGBA_MAKE

```
RGBA_MAKE(r, g, b, a)    \
    ((D3DCOLOR) (((a) << 24) | ((r) << 16) | ((g) << 8) | (b)))
```

Creates an RGBA D3DCOLOR value from supplied red, green, blue, and alpha components.

- Returns the color.

r, *g*, *b*, and *a*
Red, green, blue, and alpha components of the RGBA color to be created.

RGBA_SETALPHA

```
RGBA_SETALPHA(rgba, x)    (((x) << 24) | ((rgba) & 0x00ffffff))
```

Sets the alpha component of an RGBA D3DCOLOR value.

- Returns the RGBA color whose alpha component has been set.

rgba
RGBA color for which the alpha component will be set.

x
Value of alpha component to be set.

RGBA_TORGB

```
RGBA_TORGB(rgba)     ((D3DCOLOR) ((rgba) & 0xffffff))
```

Creates an RGB D3DCOLOR value from a supplied RGBA D3DCOLOR value by stripping off the alpha component of the color.

- Returns the RGB color.

rgba
RGBA color to be converted to an RGB color.

D3DENUMDEVICESCALLBACK

```
typedef HRESULT (FAR PASCAL * LPD3DENUMDEVICESCALLBACK)
    (LPGUID lpGuid, LPSTR lpDeviceDescription,
    LPSTR lpDeviceName, LPD3DDEVICEDESC lpD3DHWDeviceDesc,
    LPD3DDEVICEDESC lpD3DHELDeviceDesc, LPVOID lpUserArg);
```

Prototype definition for the callback function to enumerate installed Direct3D devices.

- Applications should return one of the following values:

  D3DENUMRET_CANCEL

  Cancel the enumeration.

  D3DENUMRET_OK

  Continue the enumeration.

lpGuid

Address of a globally unique identifier (GUID).

lpDeviceDescription

Address of a textual description of the device.

lpDeviceName

Address of the device name.

lpD3DHWDeviceDesc

Address containing the hardware capabilities of the Direct3D device.

lpD3DHELDeviceDesc

Address containing the emulated capabilities of the Direct3D device.

lpUserArg

Address of application-defined data passed to this callback function.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DENUMTEXTUREFORMATSCALLBACK

```
typedef HRESULT (WINAPI* LPD3DENUMTEXTUREFORMATSCALLBACK)
    (LPDDSURFACEDESC lpDdsd, LPVOID lpUserArg);
```

Prototype definition for the callback function to enumerate texture formats.

lpDdsd

Address of the DirectDrawSurface object containing the texture information.

lpUserArg

Address of application-defined data passed to this callback function.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

D3DVALIDATECALLBACK

```
typedef HRESULT (WINAPI* LPD3DVALIDATECALLBACK)
    (LPVOID lpUserArg, DWORD dwOffset);
```

Application-defined callback function supplied when an application calls the IDirect3DExecuteBuffer::Validate method. This method is a debugging routine that checks the execute buffer and returns an offset into the buffer when any errors are encountered.

lpUserArg

Address of application-defined data passed to this callback function.

dwOffset

Offset into the execute buffer at which the system found an error.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

IDirect3D Interface Method Groups

Applications use the methods of the **IDirect3D** interface to create Direct3D objects and set up the environment. The methods can be organized into the following groups:

Creation	CreateLight
	CreateMaterial
	CreateViewport
Enumeration and initialization	EnumDevices
	FindDevice
	Initialize
IUnknown	AddRef
	QueryInterface
	Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3D object without affecting the functionality of the original interface.

IDirect3D::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3D object by 1. This method is part of the IUnknown interface inherited by Direct3D.

- Returns the new reference count of the object.

When the Direct3D object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3D::Release method to decrease the reference count of the object by 1.

IDirect3D::CreateLight

```
HRESULT CreateLight(LPDIRECT3DLIGHT* lplpDirect3Dlight,
    IUnknown* pUnkOuter);
```

Allocates a Direct3DLight object. This object can then be associated with a viewport by using the IDirect3DViewport::AddLight method.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lplpDirect3DLight
Address that will be filled with a pointer to an IDirect3DLight interface if the call succeeds.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D::CreateLight** method returns an error if this parameter is anything but NULL.

IDirect3D::CreateMaterial

```
HRESULT CreateMaterial(LPDIRECT3DMATERIAL* lplpDirect3DMaterial,
    IUnknown* pUnkOuter);
```

Allocates a Direct3DMaterial object.

- Returns D3D_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Immediate-Mode Return Values.

lplpDirect3DMaterial

Address that will be filled with a pointer to an IDirect3DMaterial interface if the call succeeds.

pUnkOuter

This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D::CreateMaterial** method returns an error if this parameter is anything but NULL.

IDirect3D::CreateViewport

```
HRESULT CreateViewport(LPDIRECT3DVIEWPORT* lplpD3DViewport,
    IUnknown* pUnkOuter);
```

Creates a Direct3DViewport object. The viewport is associated with a Direct3DDevice object by using the IDirect3DDevice::AddViewport method.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lplpD3DViewport
Address that will be filled with a pointer to an IDirect3DViewport interface if the call succeeds.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D::CreateViewport** method returns an error if this parameter is anything but NULL.

IDirect3D::EnumDevices

```
HRESULT EnumDevices(LPD3DENUMDEVICESCALLBACK lpEnumDevicesCallback,
    LPVOID lpUserArg);
```

Enumerates all Direct3D device drivers installed on the system.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpEnumDevicesCallback
Address of the D3DENUMDEVICESCALLBACK callback function that the enumeration procedure will call every time a match is found.

lpUserArg
Address of application-defined data passed to the callback function.

IDirect3D::FindDevice

```
HRESULT FindDevice(LPD3DFINDDEVICESEARCH lpD3DFDS,
    LPD3DFINDDEVICERESULT lpD3DFDR);
```

Finds a device with specified characteristics and retrieves a description of it.

- Returns D3D_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Immediate-Mode Return Values.

lpD3DFDS
Address of the D3DFINDDEVICESEARCH structure describing the device to be located.

lpD3DFDR
Address of the D3DFINDDEVICERESULT structure describing the device if it is found.

IDirect3D::Initialize

```
HRESULT Initialize(REFIID lpREFIID);
```

This method is provided for compliance with the COM protocol.

- Returns DDERR_ALREADYINITIALIZED because the Direct3D object is initialized when it is created.

lpREFIID
Address of a universally unique identifier (UUID).

IDirect3D::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3D object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3D.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3D::QueryInterface** method allows Direct3D objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3D::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3D object by 1. This method is part of the IUnknown interface inherited by Direct3D.

- Returns the new reference count of the object.

The Direct3D object deallocates itself when its reference count reaches 0. Use the IDirect3D::AddRef method to increase the reference count of the object by 1.

IDirect3DDevice Interface Method Groups

Applications use the methods of the **IDirect3DDevice** interface to retrieve and set the capabilities of Direct3D devices. The methods can be organized into the following groups:

Execution	CreateExecuteBuffer Execute
Information	EnumTextureFormats GetCaps GetDirect3D GetPickRecords GetStats
IUnknown	AddRef QueryInterface Release
Matrices	CreateMatrix DeleteMatrix GetMatrix SetMatrix
Miscellaneous	Initialize Pick SwapTextureHandles
Scenes	BeginScene EndScene
Viewports	AddViewport DeleteViewport NextViewport

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DDevice object without affecting the functionality of the original interface.

The Direct3DDevice object is obtained through the appropriate call to the IDirect3DDevice::QueryInterface method from a DirectDrawSurface object that was created as a 3D-capable surface.

IDirect3DDevice::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DDevice object by 1. This method is part of the IUnknown interface inherited by Direct3DDevice.

- Returns the new reference count of the object.

When the Direct3DDevice object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DDevice::Release method to decrease the reference count of the object by 1.

IDirect3DDevice::AddViewport

```
HRESULT AddViewport(LPDIRECT3DVIEWPORT lpDirect3DViewport);
```

Adds the specified viewport to the list of viewport objects associated with the device.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DViewport
Address of the IDirect3DViewport interface that should be associated with this Direct3DDevice object.

IDirect3DDevice::BeginScene

```
HRESULT BeginScene();
```

Begins a scene.

- Returns D3D_OK if successful or an error otherwise.

Applications must call this method before performing any rendering, and must call IDirect3DDevice::EndScene when rendering is complete.

IDirect3DDevice::CreateExecuteBuffer

```
HRESULT CreateExecuteBuffer(LPDIRECT3DEXECUTEBUFFERDESC lpDesc,
    LPDIRECT3DEXECUTEBUFFER* lplpDirect3DExecuteBuffer,
    IUnknown* pUnkOuter);
```

Allocates an execute buffer for a display list. The list may be read by hardware DMA into VRAM for processing. All display primitives in the buffer that have indices to vertices must also have those vertices in the same buffer.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDesc
Address of a D3DEXECUTEBUFFERDESC structure that describes the Direct3DExecuteBuffer object to be created. The call will fail if a buffer of at least the specified size cannot be created.

lplpDirect3DExecuteBuffer
Address of a pointer that will be filled with the address of the new Direct3DExecuteBuffer object.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, this method returns an error if this parameter is anything but NULL.

The **D3DEXECUTEBUFFERDESC** structure describes the execute buffer to be created. At a minimum, the application must specify the size required. If the application specifies DEBCAPS_VIDEO_MEMORY in the capabilities member, Direct3D will attempt to keep the execute buffer in video memory.

The application can use the IDirect3DExecuteBuffer::Lock method to request that the memory be moved. When this method returns, it will adjust the contents of the **D3DEXECUTEBUFFERDESC** structure to indicate whether the data resides in system or video memory.

IDirect3DDevice::CreateMatrix

```
HRESULT CreateMatrix(LPD3DMATRIXHANDLE lpD3DMatHandle);
```

Creates a matrix.

- Returns D3D_OK if successful, or an error otherwise, such as DDERR_INVALIDPARAMS.

lpD3DMatHandle

Address of a variable that will contain a handle to the matrix that is created. The call will fail if a buffer of at least the size of the matrix cannot be created.

IDirect3DDevice::DeleteMatrix

```
HRESULT DeleteMatrix(D3DMATRIXHANDLE d3dMatHandle);
```

Deletes a matrix handle. This matrix handle must have been created by using the IDirect3DDevice::CreateMatrix method.

- Returns D3D_OK if successful, or an error otherwise, such as DDERR_INVALIDPARAMS.

d3dMatHandle
Matrix handle to be deleted.

IDirect3DDevice::DeleteViewport

```
HRESULT DeleteViewport(LPDIRECT3DVIEWPORT lpDirect3DViewport);
```

Removes the specified viewport from the list of viewport objects associated with the device.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DViewport
Address of the Direct3DViewport object that should be disassociated with this Direct3DDevice object.

IDirect3DDevice::EndScene

```
HRESULT EndScene();
```

Ends a scene that was begun by calling the IDirect3DDevice::BeginScene method.

- Returns D3D_OK if successful, or an error otherwise.

IDirect3DDevice::EnumTextureFormats

```
HRESULT EnumTextureFormats(
    LPD3DENUMTEXTUREFORMATSCALLBACK lpd3dEnumTextureProc,
    LPVOID lpArg);
```

Queries the current driver for a list of supported texture formats.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpd3dEnumTextureProc
Address of the D3DENUMTEXTUREFORMATSCALLBACK callback function that the enumeration procedure will call for each texture format.

lpArg
Address of application-defined data passed to the callback function.

IDirect3DDevice::Execute

```
HRESULT Execute(LPDIRECT3DEXECUTEBUFFER lpDirect3DExecuteBuffer,
    LPDIRECT3DVIEWPORT lpDirect3DViewport, DWORD dwFlags);
```

Executes a buffer.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DExecuteBuffer
Address of the execute buffer to be executed.

lpDirect3DViewport
Address of the Direct3DViewport object that describes the transformation context into which the execute buffer will be rendered.

dwFlags
Flags specifying whether or not objects in the buffer should be clipped. This parameter must be one of the following values:

D3DEXECUTE_CLIPPED
Clip any primitives in the buffer that are outside or partially outside the viewport.

D3DEXECUTE_UNCLIPPED
All primitives in the buffer are contained within the viewport.

IDirect3DDevice::GetCaps

```
HRESULT GetCaps(LPD3DDEVICEDESC lpD3DHWDevDesc,
    LPD3DDEVICEDESC lpD3DHELDevDesc);
```

Retrieves the capabilities of the Direct3DDevice object.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpD3DHWDevDesc
Address of the D3DDEVICEDESC structure that will contain the hardware features of the device.

lpD3DHELDevDesc
Address of the **D3DDEVICEDESC** structure that will contain the software emulation being provided.

This method does not retrieve the capabilities of the display device. To retrieve this information, use the IDirectDraw::GetCaps method.

IDirect3DDevice::GetDirect3D

```
HRESULT GetDirect3D(LPDIRECT3D* lpD3D);
```

Retrieves the current IDirect3D interface.

- Returns D3D_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Immediate-Mode Return Values.

lpD3D
Address that will contain the interface when the method returns.

IDirect3DDevice::GetMatrix

```
HRESULT GetMatrix(D3DMATRIXHANDLE lpD3DMatHandle,
    LPD3DMATRIX lpD3DMatrix);
```

Retrieves a matrix from a matrix handle. This matrix handle must have been created by using the IDirect3DDevice::CreateMatrix method.

- Returns D3D_OK if successful, or an error otherwise, such as DDERR_INVALIDPARAMS.

lpD3DMatHandle
Address of a variable that contains the matrix to be retrieved.

lpD3DMatrix
Address of a D3DMATRIX structure that contains the matrix when the method returns.

IDirect3DDevice::GetPickRecords

```
HRESULT GetPickRecords(LPDWORD lpCount,
    LPD3DPICKRECORD lpD3DPickRec);
```

Retrieves the pick records for a device.

- Returns D3D_OK if successful or an error otherwise.

lpCount

Address of a variable that contains the number of D3DPICKRECORD structures to retrieve.

lpD3DPickRec

Address that will contain an array of **D3DPICKRECORD** structures when the method returns.

An application typically calls this method twice. In the first call, the second parameter is set to NULL, and the first parameter retrieves a count of all relevant **D3DPICKRECORD** structures. The application then allocates sufficient memory for those structures and calls the method again, specifying the newly allocated memory for the second parameter.

IDirect3DDevice::GetStats

```
HRESULT GetStats(LPD3DSTATS lpD3DStats);
```

Retrieves statistics about a device.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpD3DStats
Address of a D3DSTATS structure that will be filled with the statistics.

IDirect3DDevice::Initialize

```
HRESULT Initialize(LPDIRECT3D lpd3d, LPGUID lpGUID,
    LPD3DDEVICEDESC lpd3ddvdesc);
```

Initializes a device.

- Returns D3D_OK if successful, or an error otherwise. For a list of possible return values, see Direct3D Immediate-Mode Return Values.

lpd3d
Address of the Direct3D device to use as an initializer.

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

lpd3ddvdesc
Address of a D3DDEVICEDESC structure describing the Direct3DDevice object to be initialized.

IDirect3DDevice::NextViewport

```
HRESULT NextViewport(LPDIRECT3DVIEWPORT lpDirect3DViewport,
    LPDIRECT3DVIEWPORT* lplpDirect3DViewport, DWORD dwFlags);
```

Enumerates the viewports associated with the device.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DViewport
Address of a viewport in the list of viewports associated with this Direct3DDevice object.

lplpDirect3DViewport
Address of the next viewport in the list of viewports associated with this Direct3DDevice object.

dwFlags
Flags specifying which viewport to retrieve from the list of viewports. The default setting is D3DNEXT_NEXT.

D3DNEXT_HEAD	Retrieve the item at the beginning of the list.
D3DNEXT_NEXT	Retrieve the next item in the list.
D3DNEXT_TAIL	Retrieve the item at the end of the list.

IDirect3DDevice::Pick

```
HRESULT Pick(LPDIRECT3DEXECUTEBUFFER lpDirect3DExecuteBuffer,
    LPDIRECT3DVIEWPORT lpDirect3DViewport, DWORD dwFlags,
    LPD3DRECT lpRect);
```

Executes a buffer without performing any rendering, but returns a z-ordered list of offsets to the primitives that cover the rectangle specified by *lpRect*.

This call fails if the Direct3DExecuteBuffer object is locked.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  D3DERR_EXECUTE_LOCKED
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DExecuteBuffer
Address of an execute buffer from which the z-ordered list is retrieved.

lpDirect3DViewport
Address of a viewport in the list of viewports associated with this Direct3DDevice object.

dwFlags
No flags are currently defined for this method.

lpRect
Address of a D3DRECT structure specifying the range of device coordinates to be picked.

If the **x1** and **x2** members of the structure specified in the *lpRect* parameter are equal, and the **y1** and **y2** members are equal, a single pixel is used for picking. The coordinates are specified in device-pixel space.

All Direct3DExecuteBuffer objects must be attached to a Direct3DDevice object in order for this method to succeed.

IDirect3DDevice::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* obp);
```

Determines if the Direct3DDevice object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DDevice.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

riid

Reference identifier of the interface being requested.

obp

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DDevice::QueryInterface** method allows Direct3DDevice objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DDevice::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DDevice object by 1. This method is part of the IUnknown interface inherited by Direct3DDevice.

- Returns the new reference count of the object.

The Direct3DDevice object deallocates itself when its reference count reaches 0. Use the IDirect3DDevice::AddRef method to increase the reference count of the object by 1.

IDirect3DDevice::SetMatrix

```
HRESULT SetMatrix(D3DMATRIXHANDLE d3dMatHandle,
    LPD3DMATRIX lpD3DMatrix);
```

Applies a matrix to a matrix handle. This matrix handle must have been created by using the IDirect3DDevice::CreateMatrix method.

- Returns D3D_OK if successful, or an error otherwise, such as DDERR_INVALIDPARAMS.

d3dMatHandle

Matrix handle to be set.

lpD3DMatrix

Address of a D3DMATRIX structure that describes the matrix to be set.

Transformations inside the execute buffer include a handle of a matrix. The **IDirect3DDevice::SetMatrix** method enables an application to change this matrix without having to lock and unlock the execute buffer.

IDirect3DDevice::SwapTextureHandles

```
HRESULT SwapTextureHandles(LPDIRECT3DTEXTURE lpD3DTex1,
    LPDIRECT3DTEXTURE lpD3DTex2);
```

Swaps two texture handles.

- Returns D3D_OK if successful or an error otherwise.

lpD3DTex1 and *lpD3DTex2*
Addresses of the textures whose handles will be swapped when the method returns.

This method is useful when an application is changing all the textures in a complicated object.

IDirect3DExecuteBuffer Interface Method Groups

Your application uses the methods of the **IDirect3DExecuteBuffer** interface to set up and control a Direct3D execute buffer. The methods can be organized into the following groups:

Execute data	GetExecuteData SetExecuteData
IUnknown	AddRef QueryInterface Release
Lock and unlock	Lock Unlock
Miscellaneous	Initialize Optimize Validate

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DExecuteBuffer object without affecting the functionality of the original interface.

IDirect3DExecuteBuffer::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DExecuteBuffer object by 1. This method is part of the IUnknown interface inherited by Direct3DExecuteBuffer.

- Returns the new reference count of the object.

When the Direct3DExecuteBuffer object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DExecuteBuffer::Release method to decrease the reference count of the object by 1.

IDirect3DExecuteBuffer::GetExecuteData

```
HRESULT GetExecuteData(LPD3DEXECUTEDATA lpData);
```

Retrieves the execute data state of the Direct3DExecuteBuffer object. The execute data is used to describe the contents of the Direct3DExecuteBuffer object.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  D3DERR_EXECUTE_LOCKED
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpData
Address of a D3DEXECUTEDATA structure that will be filled with the current execute data state of the Direct3DExecuteBuffer object.

This call fails if the Direct3DExecuteBuffer object is locked.

IDirect3DExecuteBuffer::Initialize

```
HRESULT Initialize(LPDIRECT3DDEVICE lpDirect3DDevice,
    LPD3DEXECUTEBUFFERDESC lpDesc);
```

This method is provided for compliance with the COM protocol.

- Returns DDERR_ALREADYINITIALIZED because the Direct3DExecuteBuffer object is initialized when it is created.

lpDirect3DDevice
Address of the device representing the Direct3D object.

lpDesc
Address of a D3DEXECUTEBUFFERDESC structure that describes the Direct3DExecuteBuffer object to be created. The call fails if a buffer of at least the specified size cannot be created.

IDirect3DExecuteBuffer::Lock

```
HRESULT Lock(LPD3DEXECUTEBUFFERDESC lpDesc);
```

Obtains a direct pointer to the commands in the execute buffer.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  D3DERR_EXECUTE_LOCKED
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS
  DDERR_WASSTILLDRAWING

lpDesc

Address of a D3DEXECUTEBUFFERDESC structure. When the method returns, the **lpData** member will be set to point to the actual data the application has access to. This data may reside in system or video memory, and is specified by the **dwCaps** member. The application may use the **IDirect3DExecuteBuffer::Lock** method to request that Direct3D move the data between system or video memory.

This call fails if the Direct3DExecuteBuffer object is locked—that is, if another thread is accessing the buffer, or if a IDirect3DDevice::Execute method that was issued on this buffer has not yet completed.

IDirect3DExecuteBuffer::Optimize

```
HRESULT Optimize();
```

Not currently supported.

IDirect3DExecuteBuffer::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* ppvObj);
```

Determines if the Direct3DExecuteBuffer object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DExecuteBuffer.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

riid
Reference identifier of the interface being requested.

ppvObj
Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DExecuteBuffer::QueryInterface** method allows Direct3DExecuteBuffer objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

This call fails if the Direct3DExecuteBuffer object is locked.

IDirect3DExecuteBuffer::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DExecuteBuffer object by 1. This method is part of the IUnknown interface inherited by Direct3DExecuteBuffer.

- Returns the new reference count of the object.

The Direct3DExecuteBuffer object deallocates itself when its reference count reaches 0. Use the IDirect3DExecuteBuffer::AddRef method to increase the reference count of the object by 1.

This call fails if the Direct3DExecuteBuffer object is locked.

IDirect3DExecuteBuffer::SetExecuteData

```
HRESULT SetExecuteData(LPD3DEXECUTEDATA lpData);
```

Sets the execute data state of the Direct3DExecuteBuffer object. The execute data is used to describe the contents of the Direct3DExecuteBuffer object.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  D3DERR_EXECUTE_LOCKED
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpData
Address of a D3DEXECUTEDATA structure that describes the execute buffer layout.

This call fails if the Direct3DExecuteBuffer object is locked.

IDirect3DExecuteBuffer::Unlock

```
HRESULT Unlock();
```

Releases the direct pointer to the commands in the execute buffer. This must be done prior to calling the IDirect3DDevice::Execute method for the buffer.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  D3DERR_EXECUTE_NOT_LOCKED
  DDERR_INVALIDOBJECT

IDirect3DExecuteBuffer::Validate

```
HRESULT Validate(LPDWORD lpdwOffset, LPD3DVALIDATECALLBACK lpFunc,
    LPVOID lpUserArg, DWORD dwReserved);
```

Checks an execute buffer and returns an offset into the buffer when any errors are encountered. This method is a debugging routine.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpdwOffset

Address of a variable that will be filled with the offset into the execute buffer at which an error was first detected. This parameter is filled in only if NULL is specified for the *lpFunc* parameter. If a callback function is specified for *lpFunc*, the offset for each error is passed to the callback function, and the *lpdwOffset* parameter is not set.

lpFunc

Address of an application-defined D3DVALIDATECALLBACK callback function. If this parameter is NULL, checking stops when the first error is detected.

lpUserArg

Address of application-defined data passed to the callback function.

dwReserved

Reserved for future use.

The callback function specified in the *lpFunc* parameter is called whenever an error is detected in the execute buffer. The system passes to this callback function the value specified in *lpUserArg* and the offset into the execute buffer where the error was detected.

This call fails if the Direct3DExecuteBuffer object is locked.

IDirect3DLight Interface Method Groups

Applications use the methods of the **IDirect3DLight** interface to retrieve and set the capabilities of lights. The methods can be organized into the following groups:

Get and set	GetLight SetLight
Initialization	Initialize
IUnknown	AddRef QueryInterface Release

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DLight object without affecting the functionality of the original interface.

IDirect3DLight::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DLight object by 1. This method is part of the IUnknown interface inherited by Direct3DLight.

- Returns the new reference count of the object.

When the Direct3DLight object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DLight::Release method to decrease the reference count of the object by 1.

IDirect3DLight::GetLight

```
HRESULT GetLight(LPD3DLIGHT lpLight);
```

Retrieves the light information for the Direct3DLight object.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpLight
Address of a D3DLIGHT structure that will be filled with the current light data.

IDirect3DLight::Initialize

```
HRESULT Initialize(LPDIRECT3D lpDirect3D);
```

This method is provided for compliance with the COM protocol.

- Returns DDERR_ALREADYINITIALIZED because the Direct3DLight object is initialized when it is created.

lpDirect3D
Address of the Direct3D structure representing the Direct3D object.

IDirect3DLight::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* ppvObj);
```

Determines if the Direct3DLight object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DLight.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

riid

Reference identifier of the interface being requested.

ppvObj

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DLight::QueryInterface** method allows Direct3DLight objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DLight::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DLight object by 1. This method is part of the IUnknown interface inherited by Direct3DLight.

- Returns the new reference count of the object.

The Direct3DLight object deallocates itself when its reference count reaches 0. Use the IDirect3DLight::AddRef method to increase the reference count of the object by 1.

IDirect3DLight::SetLight

```
HRESULT SetLight(LPD3DLIGHT lpLight);
```

Sets the light information for the Direct3DLight object.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpLight
Address of a D3DLIGHT structure that will be used to set the current light data.

IDirect3DMaterial Interface Method Groups

Applications use the methods of the **IDirect3DMaterial** interface to retrieve and set the properties of materials. The methods can be organized into the following groups:

Color reservation	Reserve
	Unreserve
IUnknown	AddRef
	QueryInterface
	Release
Materials	GetMaterial
	SetMaterial
Miscellaneous	GetHandle
	Initialize

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DMaterial object without affecting the functionality of the original interface.

IDirect3DMaterial::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DMaterial object by 1. This method is part of the IUnknown interface inherited by Direct3DMaterial.

- Returns the new reference count of the object.

When the Direct3DMaterial object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DMaterial::Release method to decrease the reference count of the object by 1.

IDirect3DMaterial::GetHandle

```
HRESULT GetHandle(LPDIRECT3DDEVICE lpDirect3DDevice,
    LPD3DMATERIALHANDLE lpHandle);
```

Obtains the material handle for the Direct3DMaterial object. This handle is used in all Direct3D API calls where a material is to be referenced. A material can be used by only one device at a time.

If the device is destroyed, the material is disassociated from the device.

- Returns D3D_OK if successful, or DDERR_INVALIDOBJECT otherwise.

lpDirect3DDevice
Address of the Direct3DDevice object in which the material is being used.

lpHandle
Address of a variable that will be filled with the material handle corresponding to the Direct3DMaterial object.

IDirect3DMaterial::GetMaterial

```
HRESULT GetMaterial(LPD3DMATERIAL lpMat);
```

Retrieves the material data for the Direct3DMaterial object.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpMat
Address of a D3DMATERIAL structure that will be filled with the current material properties.

IDirect3DMaterial::Initialize

```
HRESULT Initialize(LPDIRECT3D lpDirect3D);
```

This method is provided for compliance with the COM protocol.

- Returns DDERR_ALREADYINITIALIZED because the Direct3DMaterial object is initialized when it is created.

lpDirect3D
Address of the Direct3D structure representing the Direct3D object.

IDirect3DMaterial::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* ppvObj);
```

Determines if the Direct3DMaterial object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DMaterial.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

riid

Reference identifier of the interface being requested.

ppvObj

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DMaterial::QueryInterface** method allows Direct3DMaterial objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DMaterial::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DMaterial object by 1. This method is part of the IUnknown interface inherited by Direct3DMaterial.

- Returns the new reference count of the object.

The Direct3DMaterial object deallocates itself when its reference count reaches 0. Use the IDirect3DMaterial::AddRef method to increase the reference count of the object by 1.

IDirect3DMaterial::Reserve

```
HRESULT Reserve();
```

Not currently implemented.

IDirect3DMaterial::SetMaterial

```
HRESULT SetMaterial(LPD3DMATERIAL lpMat);
```

Sets the material data for the Direct3DMaterial object.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpMat
Address of a D3DMATERIAL structure that contains the material properties.

IDirect3DMaterial::Unreserve

```
HRESULT Unreserve();
```

Not currently implemented.

IDirect3DTexture Interface Method Groups

Applications use the methods of the **IDirect3DTexture** interface to retrieve and set the properties of textures. The methods can be organized into the following groups:

Handles	GetHandle
Initialization	Initialize
IUnknown	AddRef QueryInterface Release
Loading	Load Unload
Palette information	PaletteChanged

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DTexture object without affecting the functionality of the original interface.

The Direct3DTexture object is obtained through the appropriate call to the IDirect3D::QueryInterface method from a DirectDrawSurface object that was created as a texture map.

IDirect3DTexture::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DTexture object by 1. This method is part of the IUnknown interface inherited by Direct3DTexture.

- Returns the new reference count of the object.

When the Direct3DTexture object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DTexture::Release method to decrease the reference count of the object by 1.

IDirect3DTexture::GetHandle

```
HRESULT GetHandle(LPDIRECT3DDEVICE lpDirect3DDevice,
    LPD3DTEXTUREHANDLE lpHandle);
```

Obtains the texture handle for the Direct3DTexture object. This handle is used in all Direct3D API calls where a texture is to be referenced.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  D3DERR_BADOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DDevice
Address of the Direct3DDevice object into which the texture is to be loaded.

lpHandle
Address that will contain the texture handle corresponding to the Direct3DTexture object.

IDirect3DTexture::Initialize

```
HRESULT Initialize(LPDIRECT3DDEVICE lpD3DDevice,
    LPDIRECTDRAWSURFACE lpDDSurface);
```

This method is provided for compliance with the COM protocol.

- Returns DDERR_ALREADYINITIALIZED because the Direct3DTexture object is initialized when it is created.

lpD3DDevice
Address of the device representing the Direct3D object.

lpDDSurface
Address of the DirectDraw surface for this object.

IDirect3DTexture::Load

```
HRESULT Load(LPDIRECT3DTEXTURE lpD3DTexture);
```

Loads a texture that was created with the DDSCAPS_ALLOCONLOAD flag, which indicates that memory for the DirectDraw surface is not allocated until the surface is loaded by using this method.

- Returns D3D_OK if successful, or an error otherwise. For a list of possible return values, see Direct3D Immediate-Mode Return Values.

lpD3DTexture
Address of the texture to load.

IDirect3DTexture::PaletteChanged

```
HRESULT PaletteChanged(DWORD dwStart, DWORD dwCount);
```

Informs the driver that the palette has changed on a surface.

- Returns D3D_OK if successful, or an error otherwise. For a list of possible return values, see Direct3D Immediate-Mode Return Values.

dwStart

Index of first palette entry that has changed.

dwCount

Number of palette entries that have changed.

This method is particularly useful for applications that play video clips and therefore require palette-changing capabilities.

IDirect3DTexture::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* ppvObj);
```

Determines if the Direct3DTexture object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DTexture.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

riid

Reference identifier of the interface being requested.

ppvObj

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DTexture::QueryInterface** method allows Direct3DTexture objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DTexture::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DTexture object by 1. This method is part of the IUnknown interface inherited by Direct3DTexture.

- Returns the new reference count of the object.

The Direct3DTexture object deallocates itself when its reference count reaches 0. Use the IDirect3DTexture::AddRef method to increase the reference count of the object by 1.

IDirect3DTexture::Unload

```
HRESULT Unload();
```

Unloads the current texture.

- Returns D3D_OK if successful, or an error otherwise. For a list of possible return values, see Direct3D Immediate-Mode Return Values.

IDirect3DViewport Interface Method Groups

Applications use the methods of the **IDirect3DViewport** interface to retrieve and set the properties of viewports. The methods can be organized into the following groups:

Backgrounds	GetBackground GetBackgroundDepth SetBackground SetBackgroundDepth
Initialization	Initialize
IUnknown	AddRef QueryInterface Release
Lights	AddLight DeleteLight LightElements NextLight
Materials and viewports	Clear GetViewport SetViewport
Transformation	TransformVertices

All COM interfaces inherit the IUnknown interface methods, which are listed in the "IUnknown" group above. These three methods allow additional interfaces to be added to the Direct3DViewport object without affecting the functionality of the original interface.

IDirect3DViewport::AddLight

```
HRESULT AddLight(LPDIRECT3DLIGHT lpDirect3DLight);
```

Adds the specified light to the list of Direct3DLight objects associated with this viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DLight
Address of the Direct3DLight object that should be associated with this Direct3DDevice object.

IDirect3DViewport::AddRef

```
ULONG AddRef();
```

Increases the reference count of the Direct3DViewport object by 1. This method is part of the IUnknown interface inherited by Direct3DViewport.

- Returns the new reference count of the object.

When the Direct3DViewport object is created, its reference count is set to 1. Every time an application obtains an interface to the object or calls the **AddRef** method, the object's reference count is increased by 1. Use the IDirect3DViewport::Release method to decrease the reference count of the object by 1.

IDirect3DViewport::Clear

```
HRESULT Clear(DWORD dwCount, LPD3DRECT lpRects, DWORD dwFlags);
```

Clears the viewport or a set of rectangles in the viewport to the current background material.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

dwCount
Number of rectangles pointed to by *lpRects*.

lpRects
Address of an array of D3DRECT structures.

dwFlags
Flags indicating what to clear: the rendering target, the z-buffer, or both.

D3DCLEAR_TARGET	Clear the rendering target to the background material (if set).
D3DCLEAR_ZBUFFER	Clear the z-buffer or set it to the current background depth field (if set).

IDirect3DViewport::DeleteLight

```
HRESULT DeleteLight(LPDIRECT3DLIGHT lpDirect3DLight);
```

Removes the specified light from the list of Direct3DLight objects associated with this viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DLight
Address of the Direct3DLight object that should be disassociated with this Direct3DDevice object.

IDirect3DViewport::GetBackground

```
HRESULT GetBackground(LPD3DMATERIALHANDLE lphMat, LPBOOL lpValid);
```

Retrieves the handle of a material that represents the current background associated with the viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lphMat
Address that will contain the handle of the material being used as the background.

lpValid
Address of a variable that will be filled to indicate whether a background is associated with the viewport. If this parameter is FALSE, no background is associated with the viewport.

IDirect3DViewport::GetBackgroundDepth

```
HRESULT GetBackgroundDepth(LPDIRECTDRAWSURFACE* lplpDDSurface,
    LPBOOL lpValid);
```

Retrieves a DirectDraw surface that represents the current background-depth field associated with the viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lplpDDSurface
Address that will be initialized to point to a DirectDrawSurface object representing the background depth.

lpValid
Address of a variable that is set to FALSE if no background depth is associated with the viewport.

IDirect3DViewport::GetViewport

```
HRESULT GetViewport(LPD3DVIEWPORT lpData);
```

Retrieves the viewport registers of the viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpData
Address of a D3DVIEWPORT structure representing the viewport.

IDirect3DViewport::Initialize

```
HRESULT Initialize(LPDIRECT3D lpDirect3D);
```

This method is provided for compliance with the COM protocol.

- Returns DDERR_ALREADYINITIALIZED because the Direct3DViewport object is initialized when it is created.

lpDirect3D
Address of the Direct3D structure representing the Direct3D object.

IDirect3DViewport::LightElements

```
HRESULT LightElements(DWORD dwElementCount, LPD3DLIGHTDATA lpData);
```

Calculates light intensities and colors for rendering a geometry.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

dwElementCount
Number of elements to be lit.

lpData
Address of a D3DLIGHTDATA structure that contains the points to be lit and the resulting colors.

IDirect3DViewport::NextLight

```
HRESULT NextLight(LPDIRECT3DLIGHT lpDirect3DLight,
    LPDIRECT3DLIGHT* lplpDirect3DLight, DWORD dwFlags);
```

Enumerates the Direct3DLight objects associated with the viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDirect3DLight
Address of a light in the list of lights associated with this Direct3DDevice object.

lplpDirect3DLight
Address of a pointer that will contain the requested light in the list of lights associated with this Direct3DDevice object. The requested light is specified in the *dwFlags* parameter.

dwFlags
Flags specifying which light to retrieve from the list of lights. The default setting is D3DNEXT_NEXT.

D3DNEXT_HEAD	Retrieve the item at the beginning of the list.
D3DNEXT_NEXT	Retrieve the next item in the list.
D3DNEXT_TAIL	Retrieve the item at the end of the list.

IDirect3DViewport::QueryInterface

```
HRESULT QueryInterface(REFIID riid, LPVOID* ppvObj);
```

Determines if the Direct3DViewport object supports a particular COM interface. If it does, the system increases the reference count for the object, and the application can begin using that interface immediately. This method is part of the IUnknown interface inherited by Direct3DViewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

riid

Reference identifier of the interface being requested.

ppvObj

Address of a pointer that will be filled with the interface pointer if the query is successful.

If the application does not need to use the interface retrieved by a call to this method, it must call the **Release** method for that interface to free it. The **IDirect3DViewport::QueryInterface** method allows Direct3DViewport objects to be extended by Microsoft and third parties without interfering with each other's existing or future functionality.

IDirect3DViewport::Release

```
ULONG Release();
```

Decreases the reference count of the Direct3DViewport object by 1. This method is part of the IUnknown interface inherited by Direct3DViewport.

- Returns the new reference count of the object.

The Direct3DViewport object deallocates itself when its reference count reaches 0. Use the IDirect3DViewport::AddRef method to increase the reference count of the object by 1.

IDirect3DViewport::SetBackground

```
HRESULT SetBackground(D3DMATERIALHANDLE hMat);
```

Sets the background associated with the viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

hMat
Material handle that will be used as the background.

IDirect3DViewport::SetBackgroundDepth

```
HRESULT SetBackgroundDepth(LPDIRECTDRAWSURFACE lpDDSurface);
```

Sets the background-depth field for the viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpDDSurface
Address of the DirectDrawSurface object representing the background depth.

The z-buffer is filled with the specified depth field when the IDirect3DViewport::Clear method is called and the D3DCLEAR_ZBUFFER flag is specified. The bit depth must be 16 bits.

IDirect3DViewport::SetViewport

```
HRESULT SetViewport(LPD3DVIEWPORT lpData);
```

Sets the viewport registers of the viewport.

- Returns D3D_OK if successful, or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

lpData
Address of a D3DVIEWPORT structure that contains the new viewport.

IDirect3DViewport::TransformVertices

```
HRESULT TransformVertices(DWORD dwVertexCount,
    LPD3DTRANSFORMDATA lpData, DWORD dwFlags, LPDWORD lpOffscreen);
```

Transforms a set of vertices by the transformation matrix.

- Returns D3D_OK if successful or an error otherwise, which may be one of the following values:
  DDERR_INVALIDOBJECT
  DDERR_INVALIDPARAMS

dwVertexCount
Number of vertices in the *lpData* parameter to be transformed.

lpData
Address of a D3DTRANSFORMDATA structure that contains the vertices to be transformed.

dwFlags
One of the following flags. See the comments section following the parameter description for a discussion of how to use these flags.
D3DTRANSFORM_CLIPPED
D3DTRANSFORM_UNCLIPPED

lpOffscreen
Address of a variable that is set to a nonzero value if the resulting vertices are all off-screen.

If the *dwFlags* parameter is set to D3DTRANSFORM_CLIPPED, this method uses the current transformation matrix to transform a set of vertices, checking the resulting vertices to see if they are within the viewing frustum. The homogeneous part of the D3DTLVERTEX structure within *lpData* will be set if the vertex is clipped; otherwise only the screen coordinates will be set. The clip intersection of all the vertices transformed is returned in *lpOffscreen*. That is, if *lpOffscreen* is nonzero, all the vertices were off-screen and not straddling the viewport. The **drExtent** member of the D3DTRANSFORMDATA structure will also be set to the 2D bounding rectangle of the resulting vertices.

If the *dwFlags* parameter is set to D3DTRANSFORM_UNCLIPPED, this method uses the current transformation matrix to transform a set of vertices. In this case, the system assumes that all the resulting coordinates will be within the viewing frustum. The **drExtent** member of the **D3DTRANSFORMDATA** structure will be set to the bounding rectangle of the resulting vertices.

The **dwClip** member of **D3DTRANSFORMDATA** can help the transformation module determine whether the geometry will need clipping against the viewing volume. Before transforming a geometry, high-level software often can test whether bounding boxes or bounding spheres are wholly within the viewing volume, allowing clipping tests to be skipped, or wholly outside the viewing volume, allowing the geometry to be skipped entirely.

D3DBRANCH

```
typedef struct _D3DBRANCH {
    DWORD dwMask;
    DWORD dwValue;
    BOOL  bNegate;
    DWORD dwOffset;
} D3DBRANCH, *LPD3DBRANCH;
```

Performs conditional operations inside an execute buffer. This structure is a forward-branch structure.

dwMask

Bitmask for the branch. This mask is combined with the driver-status mask by using the logical **AND** operator. If the result equals the value specified in the **dwValue** member and the **bNegate** member is FALSE, the branch is taken.

For a list of the available driver-status masks, see the **dwStatus** member of the D3DSTATUS structure.

dwValue

Application-defined value to compare against the operation described in the **dwMask** member.

bNegate

TRUE to negate comparison.

dwOffset

How far to branch forward. Specify zero to exit.

D3DCOLORVALUE

```
typedef struct _D3DCOLORVALUE {
    union {
        D3DVALUE r;
        D3DVALUE dvR;
    };
    union {
        D3DVALUE g;
        D3DVALUE dvG;
    };
    union {
        D3DVALUE b;
        D3DVALUE dvB;
    };
    union {
        D3DVALUE a;
        D3DVALUE dvA;
    };
} D3DCOLORVALUE;
```

Describes color values for the D3DLIGHT and D3DMATERIAL structures.

dvR, **dvG**, **dvB**, and **dvA**

Values of the D3DVALUE type specifying the red, green, blue, and alpha components of a color.

D3DDEVICEDESC

```
typedef struct _D3DDeviceDesc {
    DWORD             dwSize;
    DWORD             dwFlags;
    D3DCOLORMODEL     dcmColorModel;
    DWORD             dwDevCaps;
    D3DTRANSFORMCAPS  dtcTransformCaps;
    BOOL              bClipping;
    D3DLIGHTINGCAPS   dlcLightingCaps;
    D3DPRIMCAPS       dpcLineCaps;
    D3DPRIMCAPS       dpcTriCaps;
    DWORD             dwDeviceRenderBitDepth;
    DWORD             dwDeviceZBufferBitDepth;
    DWORD             dwMaxBufferSize;
    DWORD             dwMaxVertexCount;
} D3DDEVICEDESC, *LPD3DDEVICEDESC;
```

Contains a description of the current device. This structure is used to query the current device by such methods as IDirect3DDevice::GetCaps.

dwSize
Size, in bytes, of this structure.

dwFlags
Flags identifying the members of this structure that contain valid data.

D3DDD_BCLIPPING
The **bClipping** member is valid.

D3DDD_COLORMODEL
The **dcmColorModel** member is valid.

D3DDD_DEVCAPS
The **dwDevCaps** member is valid.

D3DDD_LIGHTINGCAPS
The **dlcLightingCaps** member is valid.

D3DDD_LINECAPS
The **dpcLineCaps** member is valid.

D3DDD_MAXBUFFERSIZE
The **dwMaxBufferSize** member is valid.

D3DDD_MAXVERTEXCOUNT
The **dwMaxVertexCount** member is valid.

D3DDD_TRANSFORMCAPS
The **dtcTransformCaps** member is valid.

D3DDD_TRICAPS
The **dpcTriCaps** member is valid.

dcmColorModel
One of the members of the D3DCOLORMODEL enumerated type, specifying the color model for the device.

dwDevCaps
Flags identifying the capabilities of the device.

D3DDEVCAPS_EXECUTESYSTEMMEMORY
Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY
Device can use execute buffers from video memory.

D3DDEVCAPS_FLOATTLVERTEX

Device accepts floating point for post-transform vertex data.

D3DDEVCAPS_SORTDECREASINGZ

Device needs data sorted for decreasing depth.

D3DDEVCAPS_SORTEXACT

Device needs data sorted exactly.

D3DDEVCAPS_SORTINCREASINGZ

Device needs data sorted for increasing depth.

D3DDEVCAPS_TEXTURESYSTEMMEMORY

Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY

Device can retrieves textures from device memory.

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY

Device can use buffers from system memory for transformed and lit vertices.

D3DDEVCAPS_TLVERTEXVIDEOMEMORY

Device can use buffers from video memory for transformed and lit vertices.

dtcTransformCaps

One of the members of the D3DTRANSFORMCAPS structure, specifying the transformation capabilities of the device.

bClipping

TRUE if the device can perform 3D clipping.

dlcLightingCaps

One of the members of the D3DLIGHTINGCAPS structure, specifying the lighting capabilities of the device.

dpcLineCaps and **dpcTriCaps**

D3DPRIMCAPS structures defining the device's support for line-drawing and triangle primitives.

dwDeviceRenderBitDepth

Device's rendering bit-depth. This can be one of the following DirectDraw bit-depth constants: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

dwDeviceZBufferBitDepth

Device's z-buffer bit-depth. This can be one of the following DirectDraw bit-depth constants: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

dwMaxBufferSize

Maximum size of the execute buffer for this device. If this member is 0, the application can use any size.

dwMaxVertexCount

Maximum vertex count for this device.

D3DEXECUTEBUFFERDESC

```
typedef struct _D3DExecuteBufferDesc {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwCaps;
    DWORD  dwBufferSize;
    LPVOID lpData;
} D3DEXECUTEBUFFERDESC;
typedef D3DEXECUTEBUFFERDESC *LPD3DEXECUTEBUFFERDESC;
```

Describes the execute buffer for such methods as IDirect3DDevice::CreateExecuteBuffer and IDirect3DExecuteBuffer::Lock.

dwSize
Size of this structure, in bytes.

dwFlags
Flags identifying the members of this structure that contain valid data.

D3DDEB_BUFSIZE	The **dwBufferSize** member is valid.
D3DDEB_CAPS	The **dwCaps** member is valid.
D3DDEB_LPDATA	The **lpData** member is valid.

dwCaps
Location in memory of the execute buffer.

D3DDEBCAPS_SYSTEMMEMORY
The execute buffer data resides in system memory.

D3DDEBCAPS_VIDEOMEMORY
The execute buffer data resides in device memory.

D3DDEBCAPS_MEM
A logical **OR** of D3DDEBCAPS_SYSTEMMEMORY and D3DDEBCAPS_VIDEOMEMORY.

dwBufferSize
Size of the execute buffer, in bytes.

lpData
Address of the buffer data.

D3DEXECUTEDATA

```
typedef struct _D3DEXECUTEDATA {
    DWORD     dwSize;
    DWORD     dwVertexOffset;
    DWORD     dwVertexCount;
    DWORD     dwInstructionOffset;
    DWORD     dwInstructionLength;
    DWORD     dwHVertexOffset;
    D3DSTATUS dsStatus;
} D3DEXECUTEDATA, *LPD3DEXECUTEDATA;
```

Specifies data for the IDirect3DDevice::Execute method. When this method is called and the transformation has been done, the instruction list starting at the value specified in the **dwInstructionOffset** member is parsed and rendered.

dwSize
Size of this structure, in bytes.

dwVertexOffset
Offset into the list of vertices.

dwVertexCount
Number of vertices to execute.

dwInstructionOffset
Offset into the list of instructions to execute.

dwInstructionLength
Length of the instructions to execute.

dwHVertexOffset
Offset into the list of vertices for the homogeneous vertex used when the application is supplying screen coordinate data that needs clipping.

dsStatus
Value storing the screen extent of the rendered geometry for use after the transformation is complete. This value is a D3DSTATUS structure.

D3DFINDDEVICERESULT

```
typedef struct _D3DFINDDEVICERESULT {
    DWORD         dwSize;
    GUID          guid;
    D3DDEVICEDESC ddHwDesc;
    D3DDEVICEDESC ddSwDesc;
} D3DFINDDEVICERESULT, *LPD3DFINDDEVICERESULT;
```

Identifies a device an application has found by calling the IDirect3D::FindDevice method.

dwSize
Size, in bytes, of the structure.

guid
Globally unique identifier (GUID) of the device that was found.

ddHwDesc and **ddSwDesc**
D3DDEVICEDESC structures describing the hardware and software devices that were found.

D3DFINDDEVICESEARCH

```
typedef struct _D3DFINDDEVICESEARCH {
    DWORD         dwSize;
    DWORD         dwFlags;
    BOOL          bHardware;
    D3DCOLORMODEL dcmColorModel;
    GUID          guid;
    DWORD         dwCaps;
    D3DPRIMCAPS   dpcPrimCaps;
} D3DFINDDEVICESEARCH, *LPD3DFINDDEVICESEARCH;
```

Specifies the characteristics of a device an application wants to find. This structure is used in calls to the IDirect3D::FindDevice method.

dwSize

Size, in bytes, of this structure.

dwFlags

Flags defining the type of device the application wants to find. This member can be one or more of the following values:

D3DFDS_ALPHACMPCAPS

Match the **dwAlphaCmpCaps** member of the D3DPRIMCAPS structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_COLORMODEL

Match the color model specified in the **dcmColorModel** member of this structure.

D3DFDS_DSTBLENDCAPS

Match the **dwDestBlendCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_GUID

Match the globally unique identifier (GUID) specified in the **guid** member of this structure.

D3DFDS_HARDWARE

Match the hardware or software search specification given in the **bHardware** member of this structure.

D3DFDS_LINES

Match the **D3DPRIMCAPS** structure specified by the **dpcLineCaps** member of the D3DDEVICEDESC structure.

D3DFDS_MISCCAPS

Match the **dwMiscCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_RASTERCAPS

Match the **dwRasterCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_SHADECAPS

Match the **dwShadeCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_SRCBLENDCAPS

Match the **dwSrcBlendCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTUREBLENDCAPS

Match the **dwTextureBlendCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTURECAPS

Match the **dwTextureCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTUREFILTERCAPS

Match the **dwTextureFilterCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TRIANGLES

Match the **D3DPRIMCAPS** structure specified by the **dpcTriCaps** member of the **D3DDEVICEDESC** structure.

D3DFDS_ZCMPCAPS

Match the **dwZCmpCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

bHardware

Flag specifying whether the device to find is implemented as hardware or software. If this member is TRUE, the device to search for has hardware rasterization and may also provide other hardware acceleration. Applications that use this flag should set the D3DFDS_HARDWARE bit in the **dwFlags** member.

dcmColorModel

One of the members of the D3DCOLORMODEL enumerated type, specifying whether the device to find should use the ramp or RGB color model.

guid

Globally unique identifier (GUID) of the device to find.

dwCaps

Capability flags.

dpcPrimCaps

Specifies a D3DPRIMCAPS structure defining the device's capabilities for each primitive type.

D3DHVERTEX

```
typedef struct _D3DHVERTEX {
    DWORD        dwFlags;
    union {
        D3DVALUE hx;
        D3DVALUE dvHX;
    };
    union {
        D3DVALUE hy;
        D3DVALUE dvHY;
    };
    union {
        D3DVALUE hz;
        D3DVALUE dvHZ;
    };
} D3DHVERTEX, *LPD3DHVERTEX;
```

Defines a homogeneous vertex used when the application is supplying screen coordinate data that needs clipping. This structure is part of the D3DTRANSFORMDATA structure.

dwFlags

Flags defining the clip status of the homogeneous vertex. This member can be one or more of the flags described in the **dwClip** member of the **D3DTRANSFORMDATA** structure.

dvHX, **dvHY**, and **dvHZ**

Values of the D3DVALUE type describing transformed homogeneous coordinates. These coordinates define the vertex.

D3DINSTRUCTION

```
typedef struct _D3DINSTRUCTION {
    BYTE bOpcode;
    BYTE bSize;
    WORD wCount;
} D3DINSTRUCTION, *LPD3DINSTRUCTION;
```

Defines an instruction in an execute buffer. A display list is made up from a list of variable length instructions. Each instruction begins with a common instruction header and is followed by the data required for that instruction.

bOpcode

Rendering operation, specified as a member of the D3DOPCODE enumerated type.

bSize

Size of each instruction data unit. This member can be used to skip to the next instruction in the sequence.

wCount

Number of data units of instructions that follow. This member allows efficient processing of large batches of similar instructions, such as triangles that make up a triangle mesh.

D3DLIGHT

```
typedef struct _D3DLIGHT {
    DWORD         dwSize;
    D3DLIGHTTYPE  dltType;
    D3DCOLORVALUE dcvColor;
    D3DVECTOR     dvPosition;
    D3DVECTOR     dvDirection;
    D3DVALUE      dvRange;
    D3DVALUE      dvFalloff;
    D3DVALUE      dvAttenuation0;
    D3DVALUE      dvAttenuation1;
    D3DVALUE      dvAttenuation2;
    D3DVALUE      dvTheta;
    D3DVALUE      dvPhi;
} D3DLIGHT, *LPD3DLIGHT;
```

Defines the light type in calls to methods such as IDirect3DLight::SetLight and IDirect3DLight::GetLight.

dwSize
Size, in bytes, of this structure.

dltType
Type of the light source. This value is one of the members of the D3DLIGHTTYPE enumerated type.

dcvColor
Color of the light. This member is a D3DCOLORVALUE structure.

dvPosition and **dvDirection**
Position and direction of the light in world space.

dvRange
Distance beyond which the light has no effect.

dvFalloff
Decrease in illumination between the umbra (the angle specified by the **dvTheta** member) and the outer edge of the penumbra (the angle specified by the **dvPhi** member).

dvAttenuation0
Constant light intensity. Specifies a light level that does not decrease between the light and the cutoff point given by the **dvRange** member.

dvAttenuation1
Light intensity that decreases linearly. The light intensity is 50 percent of this value halfway between the light and the cutoff point given by the **dvRange** member.

dvAttenuation2
Light intensity that decreases according to a quadratic attenuation factor.

dvTheta
Angle, in radians, of the spotlight's umbra—that is, the fully illuminated spotlight cone.

dvPhi
Angle, in radians, defining the outer edge of the spotlight's penumbra. Points outside this cone are not lit by the spotlight.

The system uses all three of the attenuation settings to determine how the effect of a light decreases with distance from the source. The following equation shows how the attentuation settings are interpreted. The value *d* here is the distance between the vertex being lit and the light:

$$f = e^{-(density \times z)}$$

For more information about lights, see Lighting Module.

D3DLIGHTDATA

```
typedef struct _D3DLIGHTDATA {
    DWORD                dwSize;
    LPD3DLIGHTINGELEMENT lpIn;
    DWORD                dwInSize;
    LPD3DTLVERTEX        lpOut;
    DWORD                dwOutSize;
} D3DLIGHTDATA, *LPD3DLIGHTDATA;
```

Describes the points to be lit and resulting colors in calls to the IDirect3DViewport::LightElements method.

dwSize
Size, in bytes, of this structure.

lpIn
Address of a D3DLIGHTINGELEMENT structure specifying the input positions and normal vectors.

dwInSize
Amount to skip from one input element to the next. This allows the application to store extra data inline with the element.

lpOut
Address of a D3DTLVERTEX structure specifying the output colors.

dwOutSize
Amount to skip from one output color to the next. This allows the application to store extra data inline with the color.

D3DLIGHTINGCAPS

```
typedef struct _D3DLIGHTINGCAPS {
    DWORD dwSize;
    DWORD dwCaps;
    DWORD dwLightingModel;
    DWORD dwNumLights;
} D3DLIGHTINGCAPS, *LPD3DLIGHTINGCAPS;
```

Describes the lighting capabilities of a device. This structure is a member of the D3DDEVICEDESC structure.

dwSize

Size, in bytes, of this structure.

dwCaps

Flags describing the capabilities of the lighting module. The following flags are defined:

D3DLIGHTCAPS_DIRECTIONAL

Directional lights are supported.

D3DLIGHTCAPS_GLSPOT

OpenGL-style spotlights are supported.

D3DLIGHTCAPS_PARALLELPOINT

Parallel point lights are supported.

D3DLIGHTCAPS_POINT

Point lights are supported.

D3DLIGHTCAPS_SPOT

Spotlights are supported.

dwLightingModel

Flags defining whether the lighting model is RGB or monochrome. The following flags are defined:

D3DLIGHTINGMODEL_MONO Monochromatic lighting model.

D3DLIGHTINGMODEL_RGB RGB lighting model.

dwNumLights

Number of lights that can be handled.

D3DLIGHTINGELEMENT

```
typedef struct _D3DLIGHTINGELEMENT {
    D3DVECTOR dvPosition;
    D3DVECTOR dvNormal;
} D3DLIGHTINGELEMENT, *LPD3DLIGHTINGELEMENT;
```

Describes the points in model space that will be lit. This structure is part of the D3DLIGHTDATA structure.

dvPosition

Value specifying the lightable point in model space. This value is a D3DVECTOR structure.

dvNormal

Value specifying the normalized unit vector. This value is a **D3DVECTOR** structure.

D3DLINE

```
typedef struct _D3DLINE {
    union {
        WORD v1;
        WORD wV1;
    };
    union {
        WORD v2;
        WORD wV2;
    };
} D3DLINE, *LPD3DLINE;
```

Describes a line for the D3DOP_LINE opcode in the **D3DOPCODE** enumerated type.

wV1 and **wV2**

Vertex indices.

Because lines are rendered by using a list of vertices that are to be joined, one less than the count of lines will be rendered.

D3DLINEPATTERN

```
typedef struct _D3DLINEPATTERN {
    WORD wRepeatFactor;
    WORD wLinePattern;
} D3DLINEPATTERN;
```

Describes a line pattern. These values are used by the D3DRENDERSTATE_LINEPATTERN render state in the **D3DRENDERSTATETYPE** enumerated type.

wRepeatFactor

Number of bits in the pattern specified by the **wLinePattern** member to use before starting over again at the beginning of the pattern.

wLinePattern

Bits specifying the line pattern. For example, the following value would produce a dotted line: 1100110011001100.

D3DLVERTEX

```
typedef struct _D3DLVERTEX {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
    DWORD        dwReserved;
    union {
        D3DCOLOR color;
        D3DCOLOR dcColor;
    };
    union {
        D3DCOLOR specular;
        D3DCOLOR dcSpecular;
    };
    union {
        D3DVALUE tu;
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv;
        D3DVALUE dvTV;
    };
} D3DLVERTEX, *LPD3DLVERTEX;
```

Defines an untransformed and lit vertex (model coordinates with color). Applications should use this structure if the hardware handles vertex transformations. This structure contains only data and a color that would be filled by software lighting.

dvX, **dvY**, and **dvZ**
Values of the D3DVALUE type specifying the homogeneous coordinates of the vertex.

dwReserved
Reserved; must be zero.

dcColor and **dcSpecular**
Values of the D3DCOLOR type specifying the color and specular component of the vertex.

dvTU and **dvTV**
Values of the **D3DVALUE** type specifying the texture coordinates of the vertex.

D3DMATERIAL

```
typedef struct _D3DMATERIAL {
    DWORD             dwSize;
    union {
        D3DCOLORVALUE diffuse;
        D3DCOLORVALUE dcvDiffuse;
    };
    union {
        D3DCOLORVALUE ambient;
        D3DCOLORVALUE dcvAmbient;
    };
    union {
        D3DCOLORVALUE specular;
        D3DCOLORVALUE dcvSpecular;
    };
    union {
        D3DCOLORVALUE emissive;
        D3DCOLORVALUE dcvEmissive;
    };
    union {
        D3DVALUE      power;
        D3DVALUE      dvPower;
    };
    D3DTEXTUREHANDLE  hTexture;
    DWORD             dwRampSize;
} D3DMATERIAL, *LPD3DMATERIAL;
```

Specifies material properties in calls to the IDirect3DMaterial::GetMaterial and IDirect3DMaterial::SetMaterial methods.

dwSize
Size, in bytes, of this structure.

dcvDiffuse, **dcvAmbient**, **dcvSpecular**, and **dcvEmissive**
Values specifying the diffuse color, ambient color, specular color, and emissive color of the material, respectively. These values are D3DCOLORVALUE structures.

dvPower
Value of the D3DVALUE type specifying the sharpness of specular highlights.

hTexture
Handle of the texture map.

dwRampSize
Size of the color ramp. For the monochromatic (ramp) driver, this value must be less than or equal to 1 for materials assigned to the background; otherwise, the background is not displayed. This behavior also occurs when a texture that is assigned to the background has an associated material whose **dwRampSize** member is greater than 1.

The texture handle is acquired from Direct3D by loading a texture into the device. The texture handle may be used only when it has been loaded into the device.

D3DMATRIX

```
typedef struct _D3DMATRIX {
    D3DVALUE _11, _12, _13, _14;
    D3DVALUE _21, _22, _23, _24;
    D3DVALUE _31, _32, _33, _34;
    D3DVALUE _41, _42, _43, _44;
} D3DMATRIX, *LPD3DMATRIX;
```

Describes a matrix for such methods as IDirect3DDevice::GetMatrix and IDirect3DDevice::SetMatrix.

D3DMATRIXLOAD

```
typedef struct _D3DMATRIXLOAD {
    D3DMATRIXHANDLE hDestMatrix;
    D3DMATRIXHANDLE hSrcMatrix;
} D3DMATRIXLOAD, *LPD3DMATRIXLOAD;
```

Describes the operand data for the D3DOP_MATRIXLOAD opcode in the **D3DOPCODE** enumerated type.

hDestMatrix and **hSrcMatrix**
Handles of the destination and source matrices. These values are D3DMATRIX structures.

D3DMATRIXMULTIPLY

```
typedef struct _D3DMATRIXMULTIPLY {
    D3DMATRIXHANDLE hDestMatrix;
    D3DMATRIXHANDLE hSrcMatrix1;
    D3DMATRIXHANDLE hSrcMatrix2;
} D3DMATRIXMULTIPLY, *LPD3DMATRIXMULTIPLY;
```

Describes the operand data for the D3DOP_MATRIXMULTIPLY opcode in the **D3DOPCODE** enumerated type.

hDestMatrix
Handle of the matrix that stores the product of the source matrices. This value is a D3DMATRIX structure.

hSrcMatrix1 and **hSrcMatrix2**
Handles of the first and second source matrices. These values are **D3DMATRIX** structures.

D3DPICKRECORD

```
typedef struct _D3DPICKRECORD {
    BYTE     bOpcode;
    BYTE     bPad;
    DWORD    dwOffset;
    D3DVALUE dvZ;
} D3DPICKRECORD, *LPD3DPICKRECORD;
```

Returns information about picked primitives in an execute buffer for the IDirect3DDevice::GetPickRecords method.

bOpcode
Opcode of the picked primitive.

bPad
Pad byte.

dwOffset
Offset from the start of the execute buffer in which the picked primitive was found.

dvZ
Depth of the picked primitive.

The x- and y-coordinates of the picked primitive are specified in the call to the IDirect3DDevice::Pick method that created the pick records.

D3DPOINT

```
typedef struct _D3DPOINT {
    WORD wCount;
    WORD wFirst;
} D3DPOINT, *LPD3DPOINT;
```

Describes operand data for the D3DOP_POINT opcode in the in **D3DOPCODE** enumerated type.

wCount
Number of points.

wFirst
Index of the first vertex.

Points are rendered by using a list of vertices.

D3DPRIMCAPS

```
typedef struct _D3DPrimCaps {
    DWORD dwSize;
    DWORD dwMiscCaps;
    DWORD dwRasterCaps;
    DWORD dwZCmpCaps;
    DWORD dwSrcBlendCaps;
    DWORD dwDestBlendCaps;
    DWORD dwAlphaCmpCaps;
    DWORD dwShadeCaps;
    DWORD dwTextureCaps;
    DWORD dwTextureFilterCaps;
    DWORD dwTextureBlendCaps;
    DWORD dwTextureAddressCaps;
    DWORD dwStippleWidth;
    DWORD dwStippleHeight;
} D3DPRIMCAPS, *LPD3DPRIMCAPS;
```

Defines the capabilities for each primitive type. This structure is used when creating a device and when querying the capabilities of a device. This structure defines several members in the D3DDEVICEDESC structure.

dwSize

Size, in bytes, of this structure.

dwMiscCaps

General capabilities for this primitive. This member can be one or more of the following:

D3DPMISCCAPS_CONFORMANT

The device conforms to the OpenGL standard.

D3DPMISCCAPS_CULLCCW

The driver supports counterclockwise culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the **D3DCULL_CCW** member of the D3DCULL enumerated type.

D3DPMISCCAPS_CULLCW

The driver supports clockwise triangle culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the **D3DCULL_CW** member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_CULLNONE

The driver does not perform triangle culling. This corresponds to the **D3DCULL_NONE** member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_LINEPATTERNREP

The driver can handle values other than 1 in the **wRepeatFactor** member of the D3DLINEPATTERN structure. (This applies only to line-drawing primitives.)

D3DPMISCCAPS_MASKPLANES

The device can perform a bitmask of color planes.

D3DPMISCCAPS_MASKZ

The device can enable and disable modification of the z-buffer on pixel operations.

dwRasterCaps

Information on raster-drawing capabilities. This member can be one or more of the following:

D3DPRASTERCAPS_DITHER

The device can dither to improve color resolution.

D3DPRASTERCAPS_FOGTABLE

The device calculates the fog value by referring to a lookup table containing fog values that are indexed to the depth of a given pixel.

D3DPRASTERCAPS_FOGVERTEX

The device calculates the fog value during the lighting operation, places the value into the alpha component of the D3DCOLOR value given for the **specular** member of the D3DTLVERTEX structure, and interpolates the fog value during rasterization.

D3DPRASTERCAPS_PAT

The driver can perform patterned drawing (lines or fills with D3DRENDERSTATE_LINEPATTERN or one of the D3DRENDERSTATE_STIPPLEPATTERN render states) for the primitive being queried.

D3DPRASTERCAPS_ROP2

The device can support raster operations other than R2_COPYPEN.

D3DPRASTERCAPS_STIPPLE

The device can stipple polygons to simulate translucency.

D3DPRASTERCAPS_SUBPIXEL

The device performs subpixel placement of z, color, and texture data, rather than working with the nearest integer pixel coordinate. This helps avoid bleed-through due to z imprecision, and jitter of color and texture values for pixels. Note that there is no corresponding state that can be enabled and disabled; the device either performs subpixel placement or it does not, and this bit is present only so that the Direct3D client will be better able to determine what the rendering quality will be.

D3DPRASTERCAPS_SUBPIXELX

The device is subpixel accurate along the x-axis only and is clamped to an integer y-axis scanline. For information about subpixel accuracy, see D3DPRASTERCAPS_SUBPIXEL.

D3DPRASTERCAPS_XOR

The device can support **XOR** operations. If this flag is not set but D3DPRIM_RASTER_ROP2 is set, then **XOR** operations must still be supported.

D3DPRASTERCAPS_ZTEST

The device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels would have been rendered.

dwZCmpCaps

Z-buffer comparison functions that the driver can perform. This member can be one or more of the following:

D3DPCMPCAPS_ALWAYS

Always pass the z test.

D3DPCMPCAPS_EQUAL

Pass the z test if the new z equals the current z.

D3DPCMPCAPS_GREATER

Pass the z test if the new z is greater than the current z.

D3DPCMPCAPS_GREATEREQUAL

Pass the z test if the new z is greater than or equal to the current z.

D3DPCMPCAPS_LESS

Pass the z test if the new z is less than the current z.

D3DPCMPCAPS_LESSEQUAL

Pass the z test if the new z is less than or equal to the current z.

D3DPCMPCAPS_NEVER

Always fail the z test.

D3DPCMPCAPS_NOTEQUAL

Pass the z test if the new z does not equal the current z.

dwSrcBlendCaps

Source blending capabilities. This member can be one or more of the following. (The RGBA values of the source and destination are indicated with the subscripts *s* and *d*.)

D3DPBLENDCAPS_BOTHINVSRCALPHA

Source blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s) and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

D3DPBLENDCAPS_BOTHSRCALPHA

Source blend factor is (A_s, A_s, A_s, A_s) and destination blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s); the destination blend selection is overridden.

D3DPBLENDCAPS_DESTALPHA

Blend factor is (A_d, A_d, A_d, A_d).

D3DPBLENDCAPS_DESTCOLOR

Blend factor is (R_d, G_d, B_d, A_d).

D3DPBLENDCAPS_INVDESTALPHA

Blend factor is (1-A_d, 1-A_d, 1-A_d, 1-A_d).

D3DPBLENDCAPS_INVDESTCOLOR

Blend factor is (1-R_d, 1-G_d, 1-B_d, 1-A_d).

D3DPBLENDCAPS_INVSRCALPHA

Blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s).

D3DPBLENDCAPS_INVSRCCOLOR

Blend factor is (1-R_d, 1-G_d, 1-B_d, 1-A_d).

D3DPBLENDCAPS_ONE

Blend factor is (1, 1, 1, 1).

D3DPBLENDCAPS_SRCALPHA

Blend factor is (A_s, A_s, A_s, A_s).

D3DPBLENDCAPS_SRCALPHASAT

Blend factor is (f, f, f, 1); f = min(A_s, 1-A_d).

D3DPBLENDCAPS_SRCCOLOR

Blend factor is (R_s, G_s, B_s, A_s).

D3DPBLENDCAPS_ZERO

Blend factor is (0, 0, 0, 0).

dwDestBlendCaps

Destination blending capabilities. This member can be the same capabilities that are defined for the **dwSrcBlendCaps** member.

dwAlphaCmpCaps

Alpha-test comparison functions that the driver can perform. This member can be the same capabilities that are defined for the **dwZCmpCaps** member.

dwShadeCaps

Shading operations that the device can perform. It is assumed, in general, that if a device supports a given command (such as D3DOP_TRIANGLE) at all, it supports the D3DSHADE_FLAT mode (as specified in the D3DSHADEMODE enumerated type). This flag specifies whether the driver can also support Gouraud and Phong shading and whether alpha color components are supported for each of the three color-generation modes. When alpha components are not supported in a given mode, the alpha value of colors generated in that mode is implicitly 255. This is the maximum possible alpha (that is, the alpha component is at full intensity).

The color, specular highlights, fog, and alpha interpolants of a triangle each have capability flags that an application can use to find out how they are implemented by the device driver. These are modified by the shade mode, color model, and by whether the alpha component of a color is blended or stippled. For more information, see Polygons.

This member can be one or more of the following:

D3DPSHADECAPS_ALPHAFLATBLEND

D3DPSHADECAPS_ALPHAFLATSTIPPLED

Device can support an alpha component for flat blended and stippled transparency,

respectively (the D3DSHADE_FLAT state for the **D3DSHADEMODE** enumerated type). In these modes, the alpha color component for a primitive is provided as part of the color for the first vertex of the primitive.

D3DPSHADECAPS_ALPHAGOURAUDBLEND

D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED

Device can support an alpha component for Gouraud blended and stippled transparency, respectively (the D3DSHADE_GOURAUD state for the **D3DSHADEMODE** enumerated type). In these modes, the alpha color component for a primitive is provided at vertices and interpolated across a face along with the other color components.

D3DPSHADECAPS_ALPHAPHONGBLEND

D3DPSHADECAPS_ALPHAPHONGSTIPPLED

Device can support an alpha component for Phong blended and stippled transparency, respectively (the D3DSHADE_PHONG state for the **D3DSHADEMODE** enumerated type). In these modes, vertex parameters are reevaluated on a per-pixel basis, applying lighting effects for the red, green, and blue color components. Phong shading is not supported for DirectX 2.

D3DPSHADECAPS_COLORFLATMONO

D3DPSHADECAPS_COLORFLATRGB

Device can support colored flat shading in the D3DCOLOR_MONO and D3DCOLOR_RGB color models, respectively. In these modes, the color component for a primitive is provided as part of the color for the first vertex of the primitive. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, of course, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORGOURAUDMONO

D3DPSHADECAPS_COLORGOURAUDRGB

Device can support colored Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided at vertices and interpolated across a face along with the other color components. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, of course, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORPHONGMONO

D3DPSHADECAPS_COLORPHONGRGB

Device can support colored Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, vertex parameters are reevaluated on a per-pixel basis. Lighting effects are applied for the red, green, and blue color components in RGB mode, and for the blue component only for monochromatic mode. Phong shading is not supported for DirectX 2.

D3DPSHADECAPS_FOGFLAT

D3DPSHADECAPS_FOGGOURAUD

D3DPSHADECAPS_FOGPHONG

Device can support fog in the flat, Gouraud, and Phong shading models, respectively. Phong shading is not supported for DirectX 2.

D3DPSHADECAPS_SPECULARFLATMONO

D3DPSHADECAPS_SPECULARFLATRGB

Device can support specular highlights in flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARGOURAUDMONO

D3DPSHADECAPS_SPECULARGOURAUDRGB

Device can support specular highlights in Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARPHONGMONO

D3DPSHADECAPS_SPECULARPHONGRGB

Device can support specular highlights in Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. Phong shading is not supported for DirectX 2.

dwTextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the following:

D3DPTEXTURECAPS_ALPHA

RGBA textures are supported in the D3DTEX_DECAL and D3DTEX_MODULATE texture filtering modes. If this capability is not set, then only RGB textures are supported in those modes. Regardless of the setting of this flag, alpha must always be supported in D3DTEX_DECAL_MASK, D3DTEX_DECAL_ALPHA, and D3DTEX_MODULATE_ALPHA filtering modes whenever those filtering modes are available.

D3DPTEXTURECAPS_BORDER

Texture mapping along borders is supported.

D3DPTEXTURECAPS_PERSPECTIVE

Perspective correction is supported.

D3DPTEXTURECAPS_POW2

All non-mipmapped textures must have widths and heights specified as powers of two if this flag is set. (Note that all mipmapped textures must always have dimensions that are powers of two.)

D3DPTEXTURECAPS_SQUAREONLY

All textures must be square.

D3DPTEXTURECAPS_TRANSPARENCY

Texture transparency is supported. (Only those texels that are not the current transparent color are drawn.)

dwTextureFilterCaps

Texture-mapping capabilities. This member can be one or more of the following:

D3DPTFILTERCAPS_LINEAR

A weighted average of a 2-by-2 area of texels surrounding the desired pixel is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DPTFILTERCAPS_LINEARMIPLINEAR

Similar to D3DPRIM_TEX_MIP_LINEAR, but interpolates between the two nearest mipmaps.

D3DPTFILTERCAPS_LINEARMIPNEAREST

Similar to D3DPRIM_TEX_MIP_NEAREST, but interpolates between the two nearest mipmaps.

D3DPTFILTERCAPS_MIPLINEAR

Similar to D3DPRIM_TEX_LINEAR, but uses the appropriate mipmap for texel selection.

D3DPTFILTERCAPS_MIPNEAREST

Similar to D3DPRIM_TEX_NEAREST, but uses the appropriate mipmap for texel selection.

D3DPTFILTERCAPS_NEAREST

The texel with coordinates nearest to the desired pixel value is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

dwTextureBlendCaps

Texture-blending capabilities. See the D3DTEXTUREBLEND enumerated type for discussions of the various texture-blending modes. This member can be one or more of the following:

D3DPTBLENDCAPS_COPY

Copy mode texture-blending (D3DTBLEND_COPY from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_DECAL

Decal texture-blending mode (D3DTBLEND_DECAL from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_DECALALPHA

Decal-alpha texture-blending mode (D3DTBLEND_DECALALPHA from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_DECALMASK

Decal-mask texture-blending mode (D3DTBLEND_DECALMASK from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_MODULATE

Modulate texture-blending mode (D3DTBLEND_MODULATE from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_MODULATEALPHA

Modulate-alpha texture-blending mode (D3DTBLEND_MODULATEALPHA from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_MODULATEMASK

Modulate-mask texture-blending mode (D3DTBLEND_MODULATEMASK from the **D3DTEXTUREBLEND** enumerated type) is supported.

dwTextureAddressCaps

Texture-addressing capabilities. This member can be one or more of the following:

D3DPTADDRESSCAPS_CLAMP

Device can clamp textures to addresses.

D3DPTADDRESSCAPS_MIRROR

Device can mirror textures to addresses.

D3DPTADDRESSCAPS_WRAP

Device can wrap textures to addresses.

dwStippleWidth and **dwStippleHeight**

Maximum width and height of the supported stipple (up to 32-by-32).

D3DPROCESSVERTICES

```
typedef struct _D3DPROCESSVERTICES {
    DWORD dwFlags;
    WORD  wStart;
    WORD  wDest;
    DWORD dwCount;
    DWORD dwReserved;
} D3DPROCESSVERTICES, *LPD3DPROCESSVERTICES;
```

Describes how vertices in the execute buffer should be handled by the driver. This is used by the D3DOP_PROCESSVERTICES opcode in the **D3DOPCODE** enumerated type.

dwFlags

One or more of the following flags indicating how the driver should process the vertices:

D3DPROCESSVERTICES_COPY

Vertices should simply be copied to the driver, because they have always been transformed and lit. If all the vertices in the execute buffer can be copied, the driver does not need to do the work of processing the vertices, and a performance improvement results.

D3DPROCESSVERTICES_NOCOLOR

Vertices should not be colored.

D3DPROCESSVERTICES_OPMASK

Specifies a bitmask of the other flags in the **dwFlags** member, exclusive of **D3DPROCESSVERTICES_NOCOLOR** and **D3DPROCESSVERTICES_UPDATEEXTENTS**.

D3DPROCESSVERTICES_TRANSFORM

Vertices should be transformed.

D3DPROCESSVERTICES_TRANSFORMLIGHT

Vertices should be transformed and lit.

D3DPROCESSVERTICES_UPDATEEXTENTS

Extents of all transformed vertices should be updated. This information is returned in the **drExtent** member of the D3DSTATUS structure.

wStart

Index of the first vertex in the source.

wDest

Index of the first vertex in the local buffer.

dwCount

Number of vertices to be processed.

dwReserved

Reserved; must be zero.

D3DRECT

```
typedef struct _D3DRECT {
    union {
        LONG x1;
        LONG lX1;
    };
    union {
        LONG y1;
        LONG lY1;
    };
    union {
        LONG x2;
        LONG lX2;
    };
    union {
        LONG y2;
        LONG lY2;
    };
} D3DRECT, *LPD3DRECT;
;
```

Rectangle definition.

lX1 and **lY1**
Coordinates of the upper-left corner of the rectangle.

lX2 and **lY2**
Coordinates of the lower-right corner of the rectangle.

D3DSPAN

```
typedef struct _D3DSPAN {
    WORD wCount;
    WORD wFirst;
} D3DSPAN, *LPD3DSPAN;
```

Defines a span for the D3DOP_SPAN opcode in the **D3DOPCODE** enumerated type. Spans join a list of points with the same y value. If the y value changes, a new span is started.

wCount
Number of spans.

wFirst
Index to first vertex.

D3DSTATE

```
typedef struct _D3DSTATE {
    union {
        D3DTRANSFORMSTATETYPE dtstTransformStateType;
        D3DLIGHTSTATETYPE     dlstLightStateType;
        D3DRENDERSTATETYPE    drstRenderStateType;
    };
    union {
        DWORD                 dwArg[1];
        D3DVALUE              dvArg[1];
    };
} D3DSTATE, *LPD3DSTATE;
```

Describes the render state for the D3DOP_STATETRANSFORM, D3DOP_STATELIGHT, and D3DOP_STATERENDER opcodes in the **D3DOPCODE** enumerated type. The first member of this structure is the relevant enumerated type and the second is the value for that type.

dtstTransformStateType, **dlstLightStateType**, and **drstRenderStateType**

One of the members of the D3DTRANSFORMSTATETYPE, D3DLIGHTSTATETYPE, or D3DRENDERSTATETYPE enumerated type specifying the render state.

dvArg

Value of the type specified in the first member of this structure.

D3DSTATS

```
typedef struct _D3DSTATS {
    DWORD dwSize;
    DWORD dwTrianglesDrawn;
    DWORD dwLinesDrawn;
    DWORD dwPointsDrawn;
    DWORD dwSpansDrawn;
    DWORD dwVerticesProcessed;
} D3DSTATS, *LPD3DSTATS;
```

Contains statistics used by the IDirect3DDevice::GetStats method.

dwSize
Size, in bytes, of this structure.

dwTrianglesDrawn, **dwLinesDrawn**, **dwPointsDrawn**, and **dwSpansDrawn**
Number of triangles, lines, points, and spans drawn since the device was created.

dwVerticesProcessed
Number of vertices processed since the device was created.

D3DSTATUS

```
typedef struct _D3DSTATUS {
    DWORD   dwFlags;
    DWORD   dwStatus;
    D3DRECT drExtent;
} D3DSTATUS, *LPD3DSTATUS;
```

Describes the current status of the execute buffer. This structure is part of the D3DEXECUTEDATA structure and is used with the D3DOP_SETSTATUS opcode in the **D3DOPCODE** enumerated type.

dwFlags

One of the following flags, specifying whether the status, the extents, or both are being set:

D3DSETSTATUS_STATUS

Set the status.

D3DSETSTATUS_EXTENTS

Set the extents specified in the **drExtent** member.

D3DSETSTATUS_ALL

Set both the status and the extents.

dwStatus

Clipping flags. This member can be one or more of the following flags:

Combination and General Flags

D3DSTATUS_CLIPINTERSECTION

Combination of all CLIPINTERSECTION flags.

D3DSTATUS_CLIPUNIONALL

Combination of all CLIPUNION flags.

D3DSTATUS_DEFAULT

Combination of D3DSTATUS_CLIPINTERSECTION and D3DSTATUS_ZNOTVISIBLE flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONLEFT

Logical **AND** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT

Logical **AND** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP

Logical **AND** of the clip flags for the vertices compared to the top of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM

Logical **AND** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT

Logical **AND** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **AND** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through **D3DSTATUS_CLIPINTERSECTIONGEN5**

Logical **AND** of the clip flags for application-defined clipping planes.

Clip Union Flags

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONBACK

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONGEN0 through **D3DSTATUS_CLIPUNIONGEN5**

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

Basic Clipping Flags

D3DCLIP_BACK

All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM

All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT

All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT

All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT

All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP

All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through **D3DCLIP_GEN5**

Application-defined clipping planes.

drExtent

A D3DRECT structure that defines a bounding box for all the relevant vertices. For example, the structure might define the area containing the output of the D3DOP_PROCESSVERTICES opcode, assuming the D3DPROCESSVERTICES_UPDATEEXTENTS flag is set in the D3DPROCESSVERTICES structure.

The status is a rolling status and is updated during each execution. The bounding box in the **drExtent** member can grow with each execution, but it does not shrink; it can be reset only by using the D3DOP_SETSTATUS opcode.

D3DTEXTURELOAD

```
typedef struct _D3DTEXTURELOAD {
    D3DTEXTUREHANDLE hDestTexture;
    D3DTEXTUREHANDLE hSrcTexture;
} D3DTEXTURELOAD, *LPD3DTEXTURELOAD;
```

Describes operand data for the D3DOP_TEXTURELOAD opcode in the **D3DOPCODE** enumerated type.

hDestTexture
Handle of the destination texture.

hSrcTexture
Handle of the source texture.

The textures referred to by the **hDestTexture** and **hSrcTexture** members must be the same size.

D3DTLVERTEX

```
typedef struct _D3DTLVERTEX {
    union {
        D3DVALUE sx;
        D3DVALUE dvSX;
    };
    union {
        D3DVALUE sy;
        D3DVALUE dvSY;
    };
    union {
        D3DVALUE sz;
        D3DVALUE dvSZ;
    };
    union {
        D3DVALUE rhw;
        D3DVALUE dvRHW;
    };
    union {
        D3DCOLOR color;
        D3DCOLOR dcColor;
    };
    union {
        D3DCOLOR specular;
        D3DCOLOR dcSpecular;
    };
    union {
        D3DVALUE tu;
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv;
        D3DVALUE dvTV;
    };
} D3DTLVERTEX, *LPD3DTLVERTEX;
```

Defines a transformed and lit vertex (screen coordinates with color) for the D3DLIGHTDATA structure.

dvSX, **dvSY**, and **dvSZ**
Values of the D3DVALUE type describing a vertex in screen coordinates.

dvRHW
Value of the **D3DVALUE** type that is the reciprocal of homogeneous w. This value is 1 divided by the distance from the origin to the object along the z-axis.

dcColor and **dcSpecular**
Values of the D3DCOLOR type describing the color and specular component of the vertex.

dvTU and **dvTV**
Values of the **D3DVALUE** type describing the texture coordinates of the vertex.

D3DTRANSFORMCAPS

```
typedef struct _D3DTransformCaps {
    DWORD dwSize;
    DWORD dwCaps;
} D3DTRANSFORMCAPS, *LPD3DTRANSFORMCAPS;
```

Describes the transformation capabilities of a device. This structure is part of the D3DDEVICEDESC structure.

dwSize

Size, in bytes, of this structure.

dwCaps

Flag specifying whether the system clips while transforming. This member can be zero or the following flag:

D3DTRANSFORMCAPS_CLIP The system clips while transforming.

D3DTRANSFORMDATA

```
typedef struct _D3DTRANSFORMDATA {
    DWORD        dwSize;
    LPVOID       lpIn;
    DWORD        dwInSize;
    LPVOID       lpOut;
    DWORD        dwOutSize;
    LPD3DHVERTEX lpHOut;
    DWORD        dwClip;
    DWORD        dwClipIntersection;
    DWORD        dwClipUnion;
    D3DRECT      drExtent;
} D3DTRANSFORMDATA, *LPD3DTRANSFORMDATA;
```

Contains information about transformations for the IDirect3DViewport::TransformVertices method.

dwSize
Size of the structure, in bytes.

lpIn
Address of the vertices to be transformed.

dwInSize
Stride of the vertices to be transformed.

lpOut
Address used to store the transformed vertices.

dwOutSize
Stride of output vertices.

lpHOut
Address of a value that contains homogeneous transformed vertices. This value is a D3DHVERTEX structure

dwClip
Flags specifying how the vertices are clipped. This member can be one or more of the following values:

D3DCLIP_BACK
Clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM
Clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT
Clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT
Clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT
Clipped by the right plane of the viewing frustum.

D3DCLIP_TOP
Clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through **D3DCLIP_GEN5**
Application-defined clipping planes.

dwClipIntersection
Flags denoting the intersection of the clip flags. This member can be one or more of the following values:

D3DSTATUS_CLIPINTERSECTIONLEFT
Logical **AND** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT

Logical **AND** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP

Logical **AND** of the clip flags for the vertices compared to the top of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM

Logical **AND** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT

Logical **AND** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **AND** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through **D3DSTATUS_CLIPINTERSECTIONGEN5**

Logical **AND** of the clip flags for application-defined clipping planes.

dwClipUnion

Flags denoting the union of the clip flags. This member can be one or more of the following values:

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONBACK

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONGEN0 through **D3DSTATUS_CLIPUNIONGEN5**

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

drExtent

Value that defines the extent of the transformed vertices. This structure is filled by the transformation module with the screen extent of the transformed geometry. For geometries that are clipped, this extent will only include vertices that are inside the viewing volume. This value is a D3DRECT structure

Each input vertex should be a three-vector vertex giving the [x y z] coordinates in model space for the geometry. The **dwInSize** member gives the amount to skip between vertices, allowing the application to store extra data inline with each vertex.

All values generated by the transformation module are stored as 16-bit precision values. The clip is treated as an integer bitfield that is set to the inclusive **OR** of the viewing volume planes that clip a given transformed vertex.

D3DTRIANGLE

```
typedef struct _D3DTRIANGLE {
    union {
        WORD v1;
        WORD wV1;
    };
    union {
        WORD v2;
        WORD wV2;
    };
    union {
        WORD v3;
        WORD wV3;
    };
    WORD     wFlags;
} D3DTRIANGLE, *LPD3DTRIANGLE;
```

Describes the base type for all triangles. The triangle is the main rendering primitive.

For related information, see the D3DOP_TRIANGLE member in the **D3DOPCODE** enumerated type.

wV1, **wV2**, and **wV3**
Vertices describing the triangle.

wFlags
Flags describing which edges of the triangle to enable. (This information is useful only in wireframe mode.) This value can be a combination of the following edge and strip and fan flags:

Edge flags

D3DTRIFLAG_EDGEENABLE1
Edge defined by **v1**–**v2**.

D3DTRIFLAG_EDGEENABLE2
Edge defined by **v2**–**v3**.

D3DTRIFLAG_EDGEENABLE3
Edge defined by **v3**–**v1**.

D3DTRIFLAG_EDGEENABLETRIANGLE
All edges.

Strip and fan flags

D3DTRIFLAG_EVEN
The **v1**–**v2** edge of the current triangle is adjacent to the **v3**–**v1** edge of the previous triangle; that is, **v1** is the previous **v1**, and **v2** is the previous **v3**.

D3DTRIFLAG_ODD
The **v1**–**v2** edge of the current triangle is adjacent to the **v2**–**v3** edge of the previous triangle; that is, **v1** is the previous **v3**, and **v2** is the previous **v2**.

D3DTRIFLAG_START
Begin the strip or fan, loading all three vertices.

D3DTRIFLAG_STARTFLAT(len)
If this triangle is culled, also cull the specified number of subsequent triangles. This length must be greater than zero and less than 30.

This structure can be used directly for all triangle fills. For flat shading, the color and specular components are taken from the first vertex. The three vertex indices **v1**, **v2**, and **v3** are vertex indexes into the vertex list at the start of the execute buffer.

Enabled edges are visible in wireframe mode. When an application displays wireframe triangles that share an edge, it typically enables only one (or neither) edge to avoid drawing the edge twice.

The D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags refer to the locations of a triangle in a

conventional triangle strip or fan. If a triangle strip had five triangles, the following flags would be used to define the strip:

D3DTRIFLAG_START
D3DTRIFLAG_ODD
D3DTRIFLAG_EVEN
D3DTRIFLAG_ODD
D3DTRIFLAG_EVEN

Similarly, the following flags would define a triangle fan with five triangles:

D3DTRIFLAG_START
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN

The following flags could define a flat triangle fan with five triangles:

D3DTRIFLAG_STARTFLAT(4)
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN

D3DVECTOR

```
typedef struct _D3DVECTOR {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
} D3DVECTOR, *LPD3DVECTOR;
```

Defines a vector for many Direct3D and Direct3DRM methods and structures.

dvX, **dvY**, and **dvZ**
Values of the D3DVALUE type describing the vector.

D3DVERTEX

```
typedef struct _D3DVERTEX {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
    union {
        D3DVALUE nx;
        D3DVALUE dvNX;
    };
    union {
        D3DVALUE ny;
        D3DVALUE dvNY;
    };
    union {
        D3DVALUE nz;
        D3DVALUE dvNZ;
    };
    union {
        D3DVALUE tu;
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv;
        D3DVALUE dvTV;
    };
} D3DVERTEX, *LPD3DVERTEX;
```

Defines an untransformed and unlit vertex (model coordinates with normal direction vector).

For related information, see the D3DOP_TRIANGLE member in the **D3DOPCODE** enumerated type.

dvX, **dvY**, and **dvZ**
Values of the D3DVALUE type describing the homogeneous coordinates of the vertex.

dvNX, **dvNY**, and **dvNZ**
Values of the **D3DVALUE** type describing the normal coordinates of the vertex.

dvTU and **dvTV**
Values of the **D3DVALUE** type describing the texture coordinates of the vertex.

D3DVIEWPORT

```
typedef struct _D3DVIEWPORT {
    DWORD    dwSize;
    DWORD    dwX;
    DWORD    dwY;
    DWORD    dwWidth;
    DWORD    dwHeight;
    D3DVALUE dvScaleX;
    D3DVALUE dvScaleY;
    D3DVALUE dvMaxX;
    D3DVALUE dvMaxY;
    D3DVALUE dvMinZ;
    D3DVALUE dvMaxZ;
} D3DVIEWPORT, *LPD3DVIEWPORT;
```

Defines the visible 3D volume and the 2D screen area that a 3D volume projects onto for the IDirect3DViewport::GetViewport and IDirect3DViewport::SetViewport methods.

When the viewport is changed, the driver builds a new transformation matrix.

The coordinates and dimensions of the viewport are given relative to the top left of the device.

dwSize
Size of this structure, in bytes.

dwX and **dwY**
Coordinates of the top-left corner of the viewport.

dwWidth and **dwHeight**
Dimensions of the viewport.

dvScaleX and **dvScaleY**
Values of the D3DVALUE type describing the scaling quantities homogeneous to screen.

dvMaxX, **dvMaxY**, **dvMinZ**, and **dvMaxZ**
Values of the **D3DVALUE** type describing the maximum and minimum homogeneous coordinates of x, y, and z.

D3DBLEND

```
typedef enum _D3DBLEND {
    D3DBLEND_ZERO            = 1,
    D3DBLEND_ONE             = 2,
    D3DBLEND_SRCCOLOR        = 3,
    D3DBLEND_INVSRCCOLOR     = 4,
    D3DBLEND_SRCALPHA        = 5,
    D3DBLEND_INVSRCALPHA     = 6,
    D3DBLEND_DESTALPHA       = 7,
    D3DBLEND_INVDESTALPHA    = 8,
    D3DBLEND_DESTCOLOR       = 9,
    D3DBLEND_INVDESTCOLOR    = 10,
    D3DBLEND_SRCALPHASAT     = 11,
    D3DBLEND_BOTHSRCALPHA    = 12,
    D3DBLEND_BOTHINVSRCALPHA = 13,
} D3DBLEND;
```

Defines the supported blend modes for the D3DRENDERSTATE_DSTBLEND values in the **D3DRENDERSTATETYPE** enumerated type. In the member descriptions that follow, the RGBA values of the source and destination are indicated with the subscripts *s* and *d*.

D3DBLEND_ZERO
Blend factor is (0, 0, 0, 0).

D3DBLEND_ONE
Blend factor is (1, 1, 1, 1).

D3DBLEND_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DBLEND_INVSRCCOLOR
Blend factor is (A_s, A_s, A_s, A_s, 1-A_s).

D3DBLEND_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DBLEND_INVSRCALPHA
Blend factor is (1-A_s, 1-A_s, 1-A_s).

D3DBLEND_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DBLEND_INVDESTALPHA
Blend factor is (1-A_d, 1-A_d, 1-A_d, 1-A_d).

D3DBLEND_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DBLEND_INVDESTCOLOR
Blend factor is (1-R_d, 1-G_d, 1-B_d, 1-A_d).

D3DBLEND_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, 1-A_d).

D3DBLEND_BOTHSRCALPHA
Source blend factor is (A_s, A_s, A_s, A_s), and destination blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s); the destination blend selection is overridden.

D3DBLEND_BOTHINVSRCALPHA
Source blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s), and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

D3DCMPFUNC

```
typedef enum _D3DCMPFUNC {
    D3DCMP_NEVER        = 1,
    D3DCMP_LESS         = 2,
    D3DCMP_EQUAL        = 3,
    D3DCMP_LESSEQUAL    = 4,
    D3DCMP_GREATER      = 5,
    D3DCMP_NOTEQUAL     = 6,
    D3DCMP_GREATEREQUAL = 7,
    D3DCMP_ALWAYS       = 8,
} D3DCMPFUNC;
```

Defines the supported compare functions for the D3DRENDERSTATE_ZFUNC and D3DRENDERSTATE_ALPHAFUNC values of the **D3DRENDERSTATETYPE** enumerated type.

D3DCMP_NEVER
Always fail the test.

D3DCMP_LESS
Accept the new pixel if its value is less than the value of the current pixel.

D3DCMP_EQUAL
Accept the new pixel if its value equals the value of the current pixel.

D3DCMP_LESSEQUAL
Accept the new pixel if its value is less than or equal to the value of the current pixel.

D3DCMP_GREATER
Accept the new pixel if its value is greater than the value of the current pixel.

D3DCMP_NOTEQUAL
Accept the new pixel if its value does not equal the value of the current pixel.

D3DCMP_GREATEREQUAL
Accept the new pixel if its value is greater than or equal to the value of the current pixel.

D3DCMP_ALWAYS
Always pass the test.

D3DCOLORMODEL

```
typedef enum _D3DCOLORMODEL {
    D3DCOLOR_MONO = 1,
    D3DCOLOR_RGB  = 2,
} D3DCOLORMODEL;
```

Defines the color model that the system will run in.

D3DCOLOR_MONO
Use a monochromatic model (or ramp model). In this model, the blue component of a vertex color is used to define the brightness of a lit vertex.

D3DCOLOR_RGB
Use a full RGB model.

D3DCULL

```
typedef enum _D3DCULL {
    D3DCULL_NONE = 1,
    D3DCULL_CW   = 2,
    D3DCULL_CCW  = 3,
} D3DCULL;
```

Defines the supported cull modes. These define how faces are culled when rendering a geometry.

D3DCULL_NONE
Do not cull faces.

D3DCULL_CW
Cull faces with clockwise vertices.

D3DCULL_CCW
Cull faces with counterclockwise vertices.

D3DFILLMODE

```
typedef enum _D3DFILLMODE {
    D3DFILL_POINT     = 1,
    D3DFILL_WIREFRAME = 2,
    D3DFILL_SOLID     = 3
} D3DFILLMODE;
```

Contains constants describing the fill mode. These values are used by the D3DRENDERSTATE_FILLMODE render state in the **D3DRENDERSTATETYPE** enumerated type.

D3DFILL_POINT
Fill points.

D3DFILL_WIREFRAME
Fill wireframes.

D3DFILL_SOLID
Fill solids.

D3DFOGMODE

```
typedef enum _D3DFOGMODE {
    D3DFOG_NONE   = 0,
    D3DFOG_EXP    = 1,
    D3DFOG_EXP2   = 2,
    D3DFOG_LINEAR = 3
} D3DFOGMODE;
```

Contains constants describing the fog mode. These values are used by the D3DRENDERSTATE_FOGTABLEMODE render state in the **D3DRENDERSTATETYPE** enumerated type.

D3DFOG_NONE
No fog effect.

D3DFOG_EXP
The fog effect intensifies exponentially, according to the following formula:

$$f = e^{-(density \times z)}$$

D3DFOG_EXP2
The fog effect intensifies exponentially with the square of the distance, according to the following formula:

$$f = e^{-(density \times z)}$$

D3DFOG_LINEAR
The fog effect intensifies linearly between the start and end points, according to the following formula:

$$f = \frac{end - z}{end - start}$$

This is the only fog mode supported for DirectX 2.
Note that fog can be considered a measure of visibility—the lower the fog value produced by one of the fog equations, the less visible an object is.

D3DLIGHTSTATETYPE

```
typedef enum _D3DLIGHTSTATETYPE {
    D3DLIGHTSTATE_MATERIAL   = 1,
    D3DLIGHTSTATE_AMBIENT    = 2,
    D3DLIGHTSTATE_COLORMODEL = 3,
    D3DLIGHTSTATE_FOGMODE    = 4,
    D3DLIGHTSTATE_FOGSTART   = 5,
    D3DLIGHTSTATE_FOGEND     = 6,
    D3DLIGHTSTATE_FOGDENSITY = 7,
} D3DLIGHTSTATETYPE;
```

Defines the light state for the D3DOP_STATELIGHT opcode. This enumerated type is part of the D3DSTATE structure.

D3DLIGHTSTATE_MATERIAL
Defines the material associated with a light. The default value is NULL.

D3DLIGHTSTATE_AMBIENT
Defines whether the light is ambient. The default value is 0.

D3DLIGHTSTATE_COLORMODEL
One of the members of the D3DCOLORMODEL enumerated type. The default value is D3DCOLOR_RGB.

D3DLIGHTSTATE_FOGMODE
One of the members of the D3DFOGMODE enumerated type. The default value is D3DFOG_NONE.

D3DLIGHTSTATE_FOGSTART
Defines the starting value for fog. The default value is 1.0.

D3DLIGHTSTATE_FOGEND
Defines the ending value for fog. The default value is 100.0.

D3DLIGHTSTATE_FOGDENSITY
Defines the density setting for fog. The default value is 1.0.

D3DLIGHTTYPE

```
typedef enum _D3DLIGHTTYPE {
    D3DLIGHT_POINT         = 1,
    D3DLIGHT_SPOT          = 2,
    D3DLIGHT_DIRECTIONAL   = 3,
    D3DLIGHT_PARALLELPOINT = 4,
    D3DLIGHT_GLSPOT        = 5,
} D3DLIGHTTYPE;
```

Defines the light type. This enumerated type is part of the D3DLIGHT structure.

D3DLIGHT_POINT
Light is a point source.

D3DLIGHT_SPOT
Light is a spotlight source.

D3DLIGHT_DIRECTIONAL
Light is a directional source.

D3DLIGHT_PARALLELPOINT
Light is a parallel source.

D3DLIGHT_GLSPOT
Light is a GL-style spotlight.

D3DOPCODE

```
typedef enum _D3DOPCODE {
    D3DOP_POINT           = 1,
    D3DOP_LINE            = 2,
    D3DOP_TRIANGLE        = 3,
    D3DOP_MATRIXLOAD      = 4,
    D3DOP_MATRIXMULTIPLY  = 5,
    D3DOP_STATETRANSFORM  = 6,
    D3DOP_STATELIGHT      = 7,
    D3DOP_STATERENDER     = 8,
    D3DOP_PROCESSVERTICES = 9,
    D3DOP_TEXTURELOAD     = 10,
    D3DOP_EXIT            = 11,
    D3DOP_BRANCHFORWARD   = 12,
    D3DOP_SPAN            = 13,
    D3DOP_SETSTATUS       = 14,
} D3DOPCODE;
```

Contains the opcodes for execute buffers.

D3DOP_POINT
Sends a point to the renderer. Operand data is described by the D3DPOINT structure.

D3DOP_LINE
Sends a line to the renderer. Operand data is described by the D3DLINE structure.

D3DOP_TRIANGLE
Sends a triangle to the renderer. Operand data is described by the D3DTRIANGLE structure.

D3DOP_MATRIXLOAD
Triggers a data transfer in the rendering engine. Operand data is described by the D3DMATRIXLOAD structure.

D3DOP_MATRIXMULTIPLY
Triggers a data transfer in the rendering engine. Operand data is described by the D3DMATRIXMULTIPLY structure.

D3DOP_STATETRANSFORM
Sets the value of internal state variables in the rendering engine for the transformation module. Operand data is a variable token and the new value. The token identifies the internal state variable, and the new value is the value to which that variable should be set. For more information about these variables, see the D3DSTATE structure and the D3DTRANSFORMSTATETYPE enumerated type.

D3DOP_STATELIGHT
Sets the value of internal state variables in the rendering engine for the lighting module. Operand data is a variable token and the new value. The token identifies the internal state variable, and the new value is the value to which that variable should be set. For more information about these variables, see the **D3DSTATE** structure and the D3DLIGHTSTATETYPE enumerated type.

D3DOP_STATERENDER
Sets the value of internal state variables in the rendering engine for the rendering module. Operand data is a variable token and the new value. The token identifies the internal state variable, and the new value is the value to which that variable should be set. For more information about these variables, see the **D3DSTATE** structure and the D3DRENDERSTATETYPE enumerated type.

D3DOP_PROCESSVERTICES
Sets both lighting and transformations for vertices. Operand data is described by the D3DPROCESSVERTICES structure.

D3DOP_TEXTURELOAD
Triggers a data transfer in the rendering engine. Operand data is described by the D3DTEXTURELOAD structure.

D3DOP_EXIT

Signals that the end of the list has been reached.

D3DOP_BRANCHFORWARD

Enables a branching mechanism within the execute buffer. For more information, see the D3DBRANCH structure.

D3DOP_SPAN

Spans a list of points with the same y value. For more information, see the D3DSPAN structure.

D3DOP_SETSTATUS

Resets the status of the execute buffer. For more information, see the D3DSTATUS structure.

An execute buffer has two parts: an array of vertices (each typically with position, normal vector, and texture coordinates) and an array of opcode/operand groups. One opcode can have several operands following it; the system simply performs the relevant operation on each operand.

D3DRENDERSTATETYPE

```
typedef enum _D3DRENDERSTATETYPE {
    D3DRENDERSTATE_TEXTUREHANDLE      = 1,
    D3DRENDERSTATE_ANTIALIAS          = 2,
    D3DRENDERSTATE_TEXTUREADDRESS     = 3,
    D3DRENDERSTATE_TEXTUREPERSPECTIVE = 4,
    D3DRENDERSTATE_WRAPU              = 5,
    D3DRENDERSTATE_WRAPV              = 6,
    D3DRENDERSTATE_ZENABLE            = 7,
    D3DRENDERSTATE_FILLMODE           = 8,
    D3DRENDERSTATE_SHADEMODE          = 9,
    D3DRENDERSTATE_LINEPATTERN        = 10,
    D3DRENDERSTATE_MONOENABLE         = 11,
    D3DRENDERSTATE_ROP2               = 12,
    D3DRENDERSTATE_PLANEMASK          = 13,
    D3DRENDERSTATE_ZWRITEENABLE       = 14,
    D3DRENDERSTATE_ALPHATESTENABLE    = 15,
    D3DRENDERSTATE_LASTPIXEL          = 16,
    D3DRENDERSTATE_TEXTUREMAG         = 17,
    D3DRENDERSTATE_TEXTUREMIN         = 18,
    D3DRENDERSTATE_SRCBLEND           = 19,
    D3DRENDERSTATE_DESTBLEND          = 20,
    D3DRENDERSTATE_TEXTUREMAPBLEND    = 21,
    D3DRENDERSTATE_CULLMODE           = 22,
    D3DRENDERSTATE_ZFUNC              = 23,
    D3DRENDERSTATE_ALPHAREF           = 24,
    D3DRENDERSTATE_ALPHAFUNC          = 25,
    D3DRENDERSTATE_DITHERENABLE       = 26,
    D3DRENDERSTATE_BLENDENABLE        = 27,
    D3DRENDERSTATE_FOGENABLE          = 28,
    D3DRENDERSTATE_SPECULARENABLE     = 29,
    D3DRENDERSTATE_ZVISIBLE           = 30,
    D3DRENDERSTATE_SUBPIXEL           = 31,
    D3DRENDERSTATE_SUBPIXELX          = 32,
    D3DRENDERSTATE_STIPPLEDALPHA      = 33,
    D3DRENDERSTATE_FOGCOLOR           = 34,
    D3DRENDERSTATE_FOGTABLEMODE       = 35,
    D3DRENDERSTATE_FOGTABLESTART      = 36,
    D3DRENDERSTATE_FOGTABLEEND        = 37,
    D3DRENDERSTATE_FOGTABLEDENSITY    = 38,
    D3DRENDERSTATE_STIPPLEENABLE      = 39,
    D3DRENDERSTATE_STIPPLEPATTERN00   = 64,
       // Stipple patterns 01 through 30 omitted here.
    D3DRENDERSTATE_STIPPLEPATTERN31   = 95,
} D3DRENDERSTATETYPE;
```

Describes the render state for the D3DOP_STATERENDER opcode. This enumerated type is part of the D3DSTATE structure. The values mentioned in the following descriptions are set in the second member of this structure.

D3DRENDERSTATE_TEXTUREHANDLE
Texture handle. The default value is NULL.

D3DRENDERSTATE_ANTIALIAS
Antialiasing primitive edges. The default value is FALSE.

D3DRENDERSTATE_TEXTUREADDRESS
One of the members of the D3DTEXTUREADDRESS enumerated type. The default value is D3DTADDRESS_WRAP.

D3DRENDERSTATE_TEXTUREPERSPECTIVE

TRUE for perspective correction. The default value is FALSE.

D3DRENDERSTATE_WRAPU

TRUE for wrapping in u direction. The default value is FALSE.

D3DRENDERSTATE_WRAPV

TRUE for wrapping in v direction. The default value is FALSE.

D3DRENDERSTATE_ZENABLE

TRUE to enable the z-buffer comparison test when writing to the frame buffer. The default value is FALSE.

D3DRENDERSTATE_FILLMODE

One or more members of the D3DFILLMODE enumerated type. The default value is D3DFILL_SOLID.

D3DRENDERSTATE_SHADEMODE

One or more members of the D3DSHADEMODE enumerated type. The default value is D3DSHADE_GOURAUD.

D3DRENDERSTATE_LINEPATTERN

The D3DLINEPATTERN structure. The default values are 0 for **wRepeatPattern** and 0 for **wLinePattern**.

D3DRENDERSTATE_ROP2

One of the 16 ROP2 binary raster operations specifying how the supplied pixels are combined with the pixels of the display surface. The default value is R2_COPYPEN. Applications can use the D3DPRASTERCAPS_ROP2 flag in the **dwRasterCaps** member of the **D3DPRIMCAPS** structure to determine whether additional raster operations are supported.

D3DRENDERSTATE_PLANEMASK

Physical plane mask whose type is **ULONG**. The default value is ~0.

D3DRENDERSTATE_ZWRITEENABLE

TRUE to enable z writes. The default value is TRUE. This member enables an application to prevent the system from updating the z-buffer with new z values.

D3DRENDERSTATE_ALPHATESTENABLE

TRUE to enable alpha tests. The default value is FALSE. This member enables applications to turn off the tests that otherwise would accept or reject a pixel based on its alpha value.

D3DRENDERSTATE_LASTPIXEL

TRUE to prevent drawing the last pixel in a line. The default value is FALSE.

D3DRENDERSTATE_TEXTUREMAG

One of the members of the D3DTEXTUREFILTER enumerated type. The default value is D3DFILTER_NEAREST.

D3DRENDERSTATE_TEXTUREMIN

One of the members of the **D3DTEXTUREFILTER** enumerated type. The default value is D3DFILTER_NEAREST.

D3DRENDERSTATE_SRCBLEND

One of the members of the D3DBLEND enumerated type. The default value is D3DBLEND_ONE.

D3DRENDERSTATE_DSTBLEND

One of the members of the **D3DBLEND** enumerated type. The default value is D3DBLEND_ZERO.

D3DRENDERSTATE_TEXTUREMAPBLEND

One of the members of the D3DTEXTUREBLEND enumerated type. The default value is D3DTBLEND_MODULATE.

D3DRENDERSTATE_CULLMODE

One of the members of the D3DCULL enumerated type. The default value is D3DCULL_CCW. Software renderers have a fixed culling order and do not support changing the culling mode.

D3DRENDERSTATE_ZFUNC

One of the members of the D3DCMPFUNC enumerated type. The default value is D3DCMP_LESSEQUAL. This member enables an application to accept or reject a pixel based on its distance from the camera.

D3DRENDERSTATE_ALPHAREF
Value specifying a reference alpha value against which pixels are tested when alpha-testing is enabled. This value's type is **D3DFIXED**. The default value is 0.

D3DRENDERSTATE_ALPHAFUNC
One of the members of the **D3DCMPFUNC** enumerated type. The default value is D3DCMP_ALWAYS. This member enables an application to accept or reject a pixel based on its alpha value.

D3DRENDERSTATE_DITHERENABLE
TRUE to enable dithering. The default value is FALSE.

D3DRENDERSTATE_BLENDENABLE
TRUE to enable alpha blending. The default value is FALSE.

D3DRENDERSTATE_FOGENABLE
TRUE to enable fog. The default value is FALSE.

D3DRENDERSTATE_SPECULARENABLE
TRUE to enable specular. The default value is TRUE.

D3DRENDERSTATE_ZVISIBLE
TRUE to enable z-checking. The default value is FALSE. Z-checking is a culling technique in which a polygon representing the screen space of an entire group of polygons is tested against the z-buffer to discover whether any of the polygons should be drawn.

D3DRENDERSTATE_SUBPIXEL
TRUE to enable subpixel correction. The default value is FALSE.

D3DRENDERSTATE_SUBPIXELX
TRUE to enable correction in X only. The default value is FALSE.

D3DRENDERSTATE_STIPPLEDALPHA
TRUE to enable stippled alpha. The default value is FALSE.

D3DRENDERSTATE_FOGCOLOR
Value whose type is D3DCOLOR. The default value is 0.

D3DRENDERSTATE_FOGTABLEMODE
One of the members of the D3DFOGMODE enumerated type. The default value is D3DFOG_NONE.

D3DRENDERSTATE_FOGTABLESTART
Fog table start. This is the position at which fog effects begin for linear fog mode.

D3DRENDERSTATE_FOGTABLEEND
Fog table end. This is the position at which fog effects reach their maximum density for linear fog mode.

D3DRENDERSTATE_FOGTABLEDENSITY
Sets the maximum fog density for linear fog mode. This value can range from 0 to 1.

D3DRENDERSTATE_STIPPLEENABLE
Enables stippling in the device driver. When stippled alpha is enabled, it must override the current stipple pattern. When stippled alpha is disabled, the stipple pattern must be returned.

D3DRENDERSTATE_STIPPLEPATTERN00 through **D3DRENDERSTATE_STIPPLEPATTERN31**
Stipple pattern. Each render state applies to a separate line of the stipple pattern.

D3DSHADEMODE

```
typedef enum _D3DSHADEMODE {
    D3DSHADE_FLAT    = 1,
    D3DSHADE_GOURAUD = 2,
    D3DSHADE_PHONG   = 3,
} D3DSHADEMODE;
```

Describes the supported shade mode for the D3DRENDERSTATE_SHADEMODE render state in the **D3DRENDERSTATETYPE** enumerated type.

D3DSHADE_FLAT
Flat shade mode. The color of the first vertex in the triangle is used to determine the color of the face.

D3DSHADE_GOURAUD
Gouraud shade mode. The color of the face is determined by a linear interpolation between all three of the triangle's vertices.

D3DSHADE_PHONG
Phong shade mode is not supported for DirectX 2.

D3DTEXTUREADDRESS

```
typedef enum _D3DTEXTUREADDRESS {
    D3DTADDRESS_WRAP   = 1,
    D3DTADDRESS_MIRROR = 2,
    D3DTADDRESS_CLAMP  = 3,
} D3DTEXTUREADDRESS;
```

Describes the supported texture address for the D3DRENDERSTATE_TEXTUREADDRESS render state in the **D3DRENDERSTATETYPE** enumerated type.

D3DTADDRESS_WRAP

The D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV render states of the **D3DRENDERSTATETYPE** enumerated type are used. This is the default setting.

D3DTADDRESS_MIRROR

Equivalent to a tiling texture-addressing mode (that is, when neither D3DRENDERSTATE_WRAPU nor D3DRENDERSTATE_WRAPV is used) except that the texture is flipped at every integer junction. For u values between 0 and 1, for example, the texture is addressed normally, between 1 and 2 the texture is flipped (mirrored), between 2 and 3 the texture is normal again, and so on.

D3DTADDRESS_CLAMP

Texture coordinates greater than 1.0 are set to 1.0, and values less than 0.0 are set to 0.0.

For more information about using the D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV render states, see Direct3D Texture Objects.

D3DTEXTUREBLEND

```
typedef enum _D3DTEXTUREBLEND {
    D3DTBLEND_DECAL         = 1,
    D3DTBLEND_MODULATE      = 2,
    D3DTBLEND_DECALALPHA    = 3,
    D3DTBLEND_MODULATEALPHA = 4,
    D3DTBLEND_DECALMASK     = 5,
    D3DTBLEND_MODULATEMASK  = 6,
    D3DTBLEND_COPY          = 7,
} D3DTEXTUREBLEND;
```

Defines the supported texture-blending modes. This enumerated type is used by the D3DRENDERSTATE_TEXTUREMAPBLEND render state in the **D3DRENDERSTATETYPE** enumerated type.

D3DTBLEND_DECAL

Decal texture-blending mode is supported. In this mode, the RGB and alpha values of the texture replace the colors that would have been used with no texturing.

D3DTBLEND_MODULATE

Modulate texture-blending mode is supported. In this mode, the RGB values of the texture are multiplied with the RGB values that would have been used with no texturing. Any alpha values in the texture replace the alpha values in the colors that would have been used with no texturing.

D3DTBLEND_DECALALPHA

Decal-alpha texture-blending mode is supported. In this mode, the RGB and alpha values of the texture are blended with the colors that would have been used with no texturing, according to the following formula:

$C = (1–A_t)C_o + A_tC_t$

In this formula, C stands for color, A for alpha, t for texture, and o for original object (before blending).

In the D3DTBLEND_DECALALPHA mode, any alpha values in the texture replace the alpha values in the colors that would have been used with no texturing.

D3DTBLEND_MODULATEALPHA

Modulate-alpha texture-blending mode is supported. In this mode, the RGB values of the texture are multiplied with the RGB values that would have been used with no texturing, and the alpha values of the texture are multiplied with the alpha values that would have been used with no texturing.

D3DTBLEND_DECALMASK

Decal-mask texture-blending mode is supported.

D3DTBLEND_MODULATEMASK

Modulate-mask texture-blending mode is supported.

D3DTBLEND_COPY

Copy texture-blending mode is supported.

Modulation combines the effects of lighting and texturing. Because colors are specified as values between and including 0 and 1, modulating (multiplying) the texture and pre-existing colors together typically produces colors that are less bright than either source. The brightness of a color component is undiminished when one of the sources for that component is white (1). The simplest way to ensure that the colors of a texture do not change when the texture is applied to an object is to ensure that the object is white (1,1,1).

D3DTEXTUREFILTER

```
typedef enum _D3DTEXTUREFILTER {
    D3DFILTER_NEAREST          = 1,
    D3DFILTER_LINEAR           = 2,
    D3DFILTER_MIPNEAREST       = 3,
    D3DFILTER_MIPLINEAR        = 4,
    D3DFILTER_LINEARMIPNEAREST = 5,
    D3DFILTER_LINEARMIPLINEAR  = 6,
} D3DTEXTUREFILTER;
```

Defines the supported texture filter modes used by the D3DRENDERSTATE_TEXTUREMAG render state in the **D3DRENDERSTATETYPE** enumerated type.

D3DFILTER_NEAREST
The texel with coordinates nearest to the desired pixel value is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DFILTER_LINEAR
A weighted average of a 2-by-2 area of texels surrounding the desired pixel is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DFILTER_MIPNEAREST
Similar to D3DFILTER_NEAREST, but uses the appropriate mipmap for texel selection.

D3DFILTER_MIPLINEAR
Similar to D3DFILTER_LINEAR, but uses the appropriate mipmap for texel selection.

D3DFILTER_LINEARMIPNEAREST
Similar to D3DFILTER_MIPNEAREST, but interpolates between the two nearest mipmaps.

D3DFILTER_LINEARMIPLINEAR
Similar to D3DFILTER_MIPLINEAR, but interpolates between the two nearest mipmaps.

D3DTRANSFORMSTATETYPE

```
typedef enum _D3DTRANSFORMSTATETYPE {
    D3DTRANSFORMSTATE_WORLD      = 1,
    D3DTRANSFORMSTATE_VIEW       = 2,
    D3DTRANSFORMSTATE_PROJECTION = 3,
} D3DTRANSFORMSTATETYPE;
```

Describes the transformation state for the D3DOP_STATETRANSFORM opcode in the **D3DOPCODE** enumerated type. This enumerated type is part of the D3DSTATE structure.

D3DTRANSFORMSTATE_WORLD
D3DTRANSFORMSTATE_VIEW
D3DTRANSFORMSTATE_PROJECTION

Define the matrices for the world, view, and projection transformations. The default values are NULL (the identity matrices).

D3DCOLOR

```
typedef DWORD D3DCOLOR, D3DCOLOR, *LPD3DCOLOR;
```

This type is the fundamental Direct3D color type.

D3DVALUE

```
typedef float D3DVALUE, *LPD3DVALUE;
```

This type is the fundamental Direct3D fractional data type.

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all Direct3D methods. See the individual method descriptions for lists of the values each can return.

D3D_OK

D3DERR_BADMAJORVERSION

D3DERR_BADMINORVERSION

D3DERR_EXECUTE_CLIPPED_FAILED

D3DERR_EXECUTE_CREATE_FAILED

D3DERR_EXECUTE_DESTROY_FAILED

D3DERR_EXECUTE_FAILED

D3DERR_EXECUTE_LOCK_FAILED

D3DERR_EXECUTE_LOCKED

D3DERR_EXECUTE_NOT_LOCKED

D3DERR_EXECUTE_UNLOCK_FAILED

D3DERR_LIGHT_SET_FAILED

D3DERR_MATERIAL_CREATE_FAILED

D3DERR_MATERIAL_DESTROY_FAILED

D3DERR_MATERIAL_GETDATA_FAILED

D3DERR_MATERIAL_SETDATA_FAILED

D3DERR_MATRIX_CREATE_FAILED

D3DERR_MATRIX_DESTROY_FAILED

D3DERR_MATRIX_GETDATA_FAILED

D3DERR_MATRIX_SETDATA_FAILED

D3DERR_SCENE_BEGIN_FAILED

D3DERR_SCENE_END_FAILED

D3DERR_SCENE_IN_SCENE

D3DERR_SCENE_NOT_IN_SCENE

D3DERR_SETVIEWPORTDATA_FAILED

D3DERR_TEXTURE_CREATE_FAILED

D3DERR_TEXTURE_DESTROY_FAILED

D3DERR_TEXTURE_GETSURF_FAILED

D3DERR_TEXTURE_LOAD_FAILED

D3DERR_TEXTURE_LOCK_FAILED

D3DERR_TEXTURE_LOCKED

D3DERR_TEXTURE_NO_SUPPORT

D3DERR_TEXTURE_NOT_LOCKED

D3DERR_TEXTURE_SWAP_FAILED

D3DERR_TEXTURE_UNLOCK_FAILED

Introduction to Joysticks

The Microsoft® DirectInput™ application programming interface (API) provides fast and consistent access to analog and digital joysticks. The DirectInput API maintains consistency with the joystick APIs of the Microsoft Win32® Software Development Kit (SDK), but has improved responsiveness and reliability by changing the device driver model. DirectInput device drivers use the registry to store settings for standard joysticks, calibration information for previously configured joysticks, and settings for OEM-supplied joysticks.

This section describes DirectInput functions, messages, and structures that support joysticks while it updates the Win32 joystick API documentation. The DirectInput APIs also apply to other ancillary input devices that track positions within an absolute coordinate system, such as touch screens, digitizing tablets, and light pens. Extended capabilities also provide support for rudder pedals, flight yokes, virtual-reality headgear, and other devices. Each device can use up to six axes of movement, a point-of-view hat, and 32 buttons.

You can design your application to use DirectInput functions to determine the capabilities of the joysticks and joystick drivers. Also, a joystick's position and button information can be processed by querying the joystick.

Joystick Capabilities

DirectInput services are loaded when the operating system is started. DirectInput supports analog and digital joysticks. Analog joysticks require real-time responsiveness and place a higher burden on the system than digital joysticks. DirectInput services can simultaneously monitor analog joysticks in a number of different configurations. These configurations range from two analog joysticks that track up to four axes of movement and use up to four buttons to four analog joysticks that track two axes of movement and use up to four buttons. In contrast, DirectInput services can support up to 16 digital joysticks, each with up to six axes of movement and up to 32 buttons.

Each axis of movement that a joystick tracks has a range of motion. The range of motion is the distance a joystick handle can move from its neutral, or resting, position to the farthest points from that resting position.

The joystick driver can support up to 16 minidrivers, with each minidriver supporting one joystick. You can retrieve the number of joysticks supported by a joystick driver by using the joyGetNumDevs function. This function returns an unsigned integer specifying the number of joysticks that the driver can support, or 0 if there is no joystick support.

Your application can determine if a joystick is attached to the computer by using the joyGetPosEx function. This function returns JOYERR_NOERROR if the specified device is attached, or JOYERR_UNPLUGGED otherwise.

Each joystick has several capabilities that are available to your application. You can retrieve the capabilities of a joystick by using the joyGetDevCaps function. This function fills a JOYCAPS structure with joystick capabilities, such as valid axes of movement for the joystick, minimum and maximum values for its coordinate system, and the number of buttons on the joystick.

Note The return value of **joyGetNumDevs** does not indicate the number of joysticks attached to the system, but the number of joysticks that the system supports.

Joystick Calibration and Testing

Microsoft Windows® 95 provides a joystick application in the Control Panel to calibrate and test joysticks, including range of motion and buttons. This application lets the user choose from a list of joysticks, including the following:

- Generic joystick configurations
- OEM joysticks
- Custom joysticks

The application allows the calibration of up to six axes of motion, 32 buttons, and a point-of-view hat for each joystick. Calibration information is stored in the registry, so the user can switch from one joystick to another without recalibrating each one every time it is used. Once the user has calibrated or selected a new joystick, the calibration application updates the registry with the selected joystick and calibration information, and notifies the joystick driver accordingly.

Additionally, this application can notify a particular joystick of changes to the registry that affect the joystick by using the joyConfigChanged function.

Joystick Position

You can query a joystick for position and button information by using the joyGetPosEx function. This function reports position information returned by the older joystick functions of the Win32 API, including position coordinates for the x-, y-, and z-axes. It also reports state information for up to four buttons. The **joyGetPosEx** function also provides access to the following information:

- Status of fourth, fifth, and sixth axes (r, u, and v)
- Rudder information
- Point-of-view hat
- State information for up to 32 buttons
- Uncalibrated (raw) joystick data
- Scaled data for a defined range of values
- Centered scaled data
- Scaled data that includes a dead zone around the neutral joystick position

Joystick Groups

This section describes the functions and structures associated with joysticks. The elements are grouped as follows:

Device capabilities	JOYCAPS joyConfigChanged joyGetDevCaps joyGetNumDevs
Querying a joystick	joyGetPosEx JOYINFOEX

joyConfigChanged

```
MMRESULT joyConfigChanged(DWORD dwFlags);
```

Notifies the joystick driver that the registry contains new joystick settings.

- Returns JOYERR_NOERROR if successful, or one of the following error values otherwise:
  JOYERR_NOCANDO JOYERR_REGISTRYNOTVALID

dwFlags

This parameter is reserved and must be set to 0.

The joystick calibration property sheet in the control panel calls this function when the user calibrates or recalibrates a joystick, or when a different joystick device is selected.

If your application is designed to customize joystick performance, such as an OEM joystick calibration application, you can use this function to notify the joystick driver that the JOYSTICK_USER values in the registry for the currently selected joystick have changed. The JOYSTICK_USER values are located in the HKEY_LOCAL_MACHINE hive of the registry.

joyGetDevCaps

```
MMRESULT joyGetDevCaps(UINT uJoyID, LPJOYCAPS pjc,
    UINT cbjc);
```

Queries a joystick to determine its capabilities.

- Returns JOYERR_NOERROR if successful, or one of the following error values otherwise:
  MMSYSERR_INVALPARAM MMSYSERR_NODRIVER

uJoyID
Identifier of the joystick, either JOYSTICKID1 or JOYSTICKID2, to be queried.

pjc
Address for a JOYCAPS structure to contain the capabilities of the joystick.

cbjc
Size of the **JOYCAPS** structure, in bytes.

You can use the joyGetNumDevs function to determine the number of joystick devices supported by the driver.

joyGetNumDevs

```
UINT joyGetNumDevs(VOID);
```

Queries the joystick driver for the number of joysticks it supports.

- Returns the number of joysticks supported by the driver, or 0 if no driver is present.

The joyGetPosEx function is used to determine whether a given joystick is physically attached to the computer.

joyGetPosEx

```
MMRESULT joyGetPosEx(UINT uJoyID, LPJOYINFOEX pji);
```

Queries a joystick for its position and button status.

- Returns JOYERR_NOERROR if successful, or one of the following error values otherwise:
  JOYERR_UNPLUGGED MMSYSERR_BADDEVICEID
  MMSYSERR_INVALPARAM MMSYSERR_NODRIVER

uJoyID
Identifier of the joystick to be queried.

pji
Address for a JOYINFOEX structure that contains extended position information and button status of the joystick.

Before calling this function, your application must identify the items to query by setting one or more flags in the **dwFlags** member of the **JOYINFOEX** structure.

This function provides access to extended devices, such as rudder pedals, point-of-view hats, devices with a large number of buttons, and coordinate systems using up to six axes.

JOYCAPS

```
typedef struct {
    WORD wMid;
    WORD wPid;
    CHAR szPname[MAXPNAMELEN];
    UINT wXmin;
    UINT wXmax;
    UINT wYmin;
    UINT wYmax;
    UINT wZmin;
    UINT wZmax;
    UINT wNumButtons;
    UINT wPeriodMin;
    UINT wPeriodMax;
\\  The following members are not in previous versions
\\  of Windows.
    UINT wRmin;
    UINT wRmax;
    UINT wUmin;
    UINT wUmax;
    UINT wVmin;
    UINT wVmax;
    UINT wCaps;
    UINT wMaxAxes;
    UINT wNumAxes;
    UINT wMaxButtons;
    CHAR szRegKey[MAXPNAMELEN];
    CHAR szOEMVxD[MAXOEMVXD];
} JOYCAPS;
```

Contains information about the specified joystick's capabilities.

wMid
Manufacturer identifier.

wPid
Product identifier.

szPname
Null-terminated string that contains the joystick product name.

wXmin and **wXmax**
Minimum and maximum x-coordinate values.

wYmin and **wYmax**
Minimum and maximum y-coordinate values.

wZmin and **wZmax**
Minimum and maximum z-coordinate values.

wNumButtons
Number of joystick buttons.

wPeriodMin and **wPeriodMax**
Minimum and maximum polling frequencies supported once an application has captured a joystick.

wRmin and **wRmax**
Minimum and maximum rudder values. The rudder is the fourth axis of movement.

wUmin and **wUmax**
Minimum and maximum u-coordinate (fifth axis) values.

wVmin and **wVmax**
Minimum and maximum v-coordinate (sixth axis) values.

wCaps

Joystick capabilities. The following flags define individual capabilities that a joystick might have:

JOYCAPS_HASPOV
The joystick has point-of-view information.

JOYCAPS_HASR
The joystick has rudder (fourth axis) information.

JOYCAPS_HASU
The joystick has u-coordinate (fifth axis) information.

JOYCAPS_HASV
The joystick has v-coordinate (sixth axis) information.

JOYCAPS_HASZ
The joystick has z-coordinate information.

JOYCAPS_POV4DIR
The joystick point-of-view supports discrete values (centered, forward, backward, left, and right).

JOYCAPS_POVCTS
The joystick point-of-view supports continuous degree bearings.

wMaxAxes
Maximum number of axes supported by the joystick.

wNumAxes
Number of axes currently in use by the joystick.

wMaxButtons
Maximum number of buttons supported by the joystick.

szRegKey
Null-terminated string that contains the registry key for the joystick.

szOEMVxD
Null-terminated string that identifies the joystick driver OEM.

JOYINFOEX

```
typedef struct joyinfoex_tag {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwXpos;
    DWORD dwYpos;
    DWORD dwZpos;
    DWORD dwRpos;
    DWORD dwUpos;
    DWORD dwVpos;
    DWORD dwButtons;
    DWORD dwButtonNumber;
    DWORD dwPOV;
    DWORD dwReserved1;
    DWORD dwReserved2;
} JOYINFOEX;
```

Contains extended information about the joystick position, the point-of-view position, and the button state.

dwSize

Size of this structure, in bytes.

dwFlags

Flags that indicate if information returned in this structure is valid. Members that do not contain valid information are set to 0. The following flags are defined:

JOY_RETURNALL

Equivalent to setting all of the JOY_RETURN values except JOY_RETURNRAWDATA.

JOY_RETURNBUTTONS

The **dwButtons** member contains valid information about the state of each joystick button.

JOY_RETURNCENTERED

Centers the joystick neutral position to the middle value of each axis of movement.

JOY_RETURNPOV

The **dwPOV** member contains valid information about the point-of-view control, which is expressed in discrete units.

JOY_RETURNPOVCTS

The **dwPOV** member contains valid information about the point-of-view control expressed in continuous, one-hundredth of a degree units.

JOY_RETURNR

The **dwRpos** member contains valid rudder pedal information. This indicates the existence of a fourth axis.

JOY_RETURNRAWDATA

Indicates that uncalibrated joystick readings are stored in this structure.

JOY_RETURNU

The **dwUpos** member contains valid data for a fifth axis of the joystick, if such an axis is available. 0 is returned otherwise.

JOY_RETURNV

The **dwVpos** member contains valid data for a sixth axis of the joystick, if such an axis is available. 0 is returned otherwise.

JOY_RETURNX

The **dwXpos** member contains valid data for the x-coordinate of the joystick.

JOY_RETURNY

The **dwYpos** member contains valid data for the y-coordinate of the joystick.

JOY_RETURNZ

The **dwZpos** member contains valid data for the z-coordinate of the joystick.

JOY_USEDEADZONE

Expands the range for the neutral position of the joystick and calls this range the dead zone. The joystick driver returns a constant value for all positions in the dead zone.

The following flags provide data to calibrate a joystick and are intended for custom calibration applications:

JOY_CAL_READ3

Reads the x-, y-, and z-coordinates, then stores the raw values in the **dwXpos**, **dwYpos**, and **dwZpos** members, respectively.

JOY_CAL_READ4

Reads the rudder information and the x-, y-, and z-coordinates, then stores the raw values in the **dwRpos**, **dwXpos**, **dwYpos**, and **dwZpos** members, respectively.

JOY_CAL_READ5

Reads the rudder information and the x-, y-, z-, and u-coordinates, then stores the raw values in the **dwRpos**, **dwXpos**, **dwYpos**, **dwZpos**, and **dwUpos** members, respectively.

JOY_CAL_READ6

Reads the raw v-axis data if a joystick minidriver is present to provide the information. Returns 0 otherwise.

JOY_CAL_READALWAYS

Reads the joystick port even if the driver does not detect a device.

JOY_CAL_READRONLY

Reads the rudder information if a joystick minidriver is present to provide the data, then stores the raw value in the **dwRpos** member. Returns 0 otherwise.

JOY_CAL_READUONLY

Reads the u-coordinate if a joystick minidriver is present to provide the data, then stores the raw value in the **dwUpos** member. Returns 0 otherwise.

JOY_CAL_READVONLY

Reads the v-coordinate if a joystick minidriver is present to provide the data, then stores the raw value in the **dwVpos** member. Returns 0 otherwise.

JOY_CAL_READXONLY

Reads the x-coordinate and stores the raw value in the **dwXpos** member.

JOY_CAL_READXYONLY

Reads the x- and y-coordinates and stores the raw values in the **dwXpos** and **dwYpos** members, respectively.

JOY_CAL_READYONLY

Reads the y-coordinate and stores the raw value in the **dwYpos** member.

JOY_CAL_READZONLY

Reads the z-coordinate and stores the raw value in the **dwZpos** member.

dwXpos, **dwYpos**, **dwZpos**

Current x-coordinate, y-coordinate, and z-coordinate positions, respectively.

dwRpos

Current position of the rudder or fourth joystick axis.

dwUpos, **dwVpos**

Current 5th axis and 6th axis positions, respectively.

dwButtons

Current state of the 32 joystick buttons. The value of this member can be set to any combination of JOY_BUTTON*n* flags, where *n* is a value ranging from one to 32. Each value corresponds to the button that is pressed.

dwButtonNumber

Current button number that is pressed.

dwPOV

Current position of the point-of-view control. Values for this member range from 0 to 35,900. These values represent each view's angle, in degrees, multiplied by 100.

dwReserved1, **dwReserved2**

Reserved, do not use.

The value of the **dwSize** member is also used to identify the version number for the structure when it is passed to the joyGetPosEx function.

Most devices with a point-of-view control have only five positions. When the JOY_RETURNPOV flag is set, these positions are reported by using the following JOY_POV constants.

Point-of-view positions	Description
JOY_POVBACKWARD	Point-of-view hat is pressed backward. The value 18,000 represents an orientation of 180.00 degrees.
JOY_POVCENTERED	Point-of-view hat is in the neutral position. The value -1 means the point-of-view hat has no angle to report.
JOY_POVFORWARD	Point-of-view hat is pressed forward. The value 0 represents an orientation of 0.00 degrees.
JOY_POVLEFT	Point-of-view hat is being pressed to the left. The value 27,000 represents an orientation of 270.00 degrees.
JOY_POVRIGHT	Point-of-view hat is pressed to the right. The value 9,000 represents an orientation of 90.00 degrees.

The default Windows 95 joystick driver currently supports these five discrete directions. If your application can accept only the defined point-of-view values, it must use the JOY_RETURNPOV flag. If your application can accept other degree readings, it should use the JOY_RETURNPOVCTS flag to obtain continuous data if it is available. The JOY_RETURNPOVCTS flag also supports the JOY_POV constants used with the JOY_RETURNPOV flag.

Return Values

Errors are represented by negative values and cannot be combined. This table lists the failures that can be returned by all DirectInput functions. For a list of the error codes each function is capable of returning, see the individual function descriptions.

JOYERR_NOCANDO

The joystick driver cannot update the device information from the registry.

JOYERR_NOERROR

The request completed successfully.

JOYERR_REGISTRYNOTVALID

One or more registry joystick entries contain invalid data.

JOYERR_UNPLUGGED

The specified joystick is not connected to the computer.

MMSYSERR_BADDEVICEID

The specified joystick identifier is invalid.

MMSYSERR_INVALPARAM

An invalid parameter was passed.

MMSYSERR_NODRIVER

The joystick driver is not present.

About DirectSetup

DirectSetup is a simple application programming interface (API) that provides you with a one-call installation for the DirectX™ 2 components. This is more than a convenience; DirectX 2 is a complex product and its installation is an involved task. You should not attempt a manual installation of DirectX 2.

Using DirectSetup

Although there are two APIs provided by DirectSetup, only one is of use to you as an application developer. The other is designed for those who plan to install their own DirectX device drivers, and is not addressed in this document.

Although DirectSetup allows you to install a single DirectX component, such as DirectDraw, this is not recommended since the space saving advantages are minimal due to the interdependent design of DirectX components.

Functions

There are only two functions provided by DirectSetup, and only one of them, DirectXSetup, is of interest to you as an application developer. The other, **DirectXDeviceSetup**, is intended for use by manufacturers who want to install their own DirectX device drivers, and is not addressed in this document. The declarations for DirectSetup can be found in Dsetup.h.

DirectXSetup

```
int WINAPI DirectXSetup(HWND hwnd, LPSTR root_path,
    DWORD dwflags);
```

Installs one or more DirectX components.

- Returns one of the following values:

  SUCCESS

  A 0 is returned if the setup was successful and *no* reboot is required.

  A 1 is returned if the setup was successful and a reboot *is* required.

  DSETUPERR_BADSOURCESIZE

  A file's size could not be verified or was incorrect.

  DSETUPERR_BADSOURCETIME

  A file's date and time could not be verified or was incorrect.

  DSETUPERR_BADWINDOWSVERSION

  The Windows version on your system is not supported by DirectX.

  DSETUPERR_CANTFINDDIR

  The setup program could not find the working directory.

  DSETUPERR_CANTFINDINF

  A required .inf file could not be found.

  DSETUPERR_INTERNAL

  An internal error occurred.

  DSETUPERR_NOCOPY

  A file's version could not be verified or was incorrect. If the DSETUP_REINSTALL flag is specified, equal versions will be installed.

  DSETUPERR_OUTOFDISKSPACE

  The setup program ran out of disk space during installation.

  DSETUPERR_SOURCEFILENOTFOUND

  One of the required source files could not be found.

hwnd

Parent HWND for the setup dialog boxes.

root_path

Address of a string containing the root path of the DirectX component files.

dwflags

One or more flags used to indicate which DirectX components should be installed. A full installation is recommended.

DSETUP_D3D	Installs Direct3D.
DSETUP_DDRAW	Installs DirectDraw.
DSETUP_DIRECTX	Installs all DirectX components.
DSETUP_DPLAY	Installs DirectPlay.
DSETUP_DSOUND	Installs DirectSound.
DSETUP_REINSTALL	Installs even if the existing components are the same version.

About AutoPlay

Microsoft® AutoPlay is a feature of the Microsoft Windows® 95 operating system. When you insert a CD containing AutoPlay into a CD-ROM drive on a Windows 95-based platform, AutoPlay automatically starts an application on the CD that installs, configures, and runs the selected product. AutoPlay automates the procedures for installing and configuring products designed for Windows 95-based platforms that are distributed on CDs.

You can use AutoPlay to install and run CD-ROM applications that run with Windows 95, whether the application is based on the MS-DOS® operating system, Windows 3.0, Windows 3.1, or Windows 95.

For your CD-ROM product to display the Microsoft Windows 95 logo, it must be enabled for AutoPlay.

Note MS-DOS and previous versions of Windows (versions 3.0 and higher) ignore AutoPlay. Adding AutoPlay to the CD does not hinder or alter user interaction when the CD-ROM is installed on a platform for MS-DOS, Windows 3.0, or Windows 3.1.

How AutoPlay Works

The implementation of AutoPlay relies on the following three items:

- 32-bit CD-ROM device drivers.

  CD-ROM device drivers for Windows 95 are designed to detect when you insert a CD into a CD-ROM drive. This feature was not present in device drivers designed for previous versions of MS-DOS and Windows. When you insert a CD into a CD-ROM drive, Windows 95 immediately checks to see if the CD has a personal computer file system. If it does, Windows 95 looks for a file named Autorun.inf.
- Autorun.inf file.

  This file resides on the CD and specifies the application that AutoPlay runs. Autorun.inf can contain other information as well.
- Startup application.

  Although you can run any application on the CD by specifying it in the Autorun.inf file, typically the application performs a startup or installation function. By including your own startup application, you can control the install, uninstall, and run processes for the program distributed on the CD.

The Autorun.inf File

The Autorun.inf file is a text file located in the root directory of the CD. This file specifies the startup application associated with the CD, as well as the icon that is displayed in Windows Explorer or after you have double-clicked My Computer. You can also specify customized menu items for the CD in Autorun.inf. These menu items are added to the shortcut menu that is displayed when the user selects the CD-ROM icon and right-clicks the mouse.

The smallest Autorun.inf file contains three lines of text and identifies the startup application and the icon for the CD-ROM:

```
[autorun]
open=filename.exe
icon=filename.ico
```

The [autorun] line identifies the block of lines that follow it as AutoPlay commands. This line is required in every Autorun.inf file. The open=*filename.exe* line specifies the path and file name of the application that AutoPlay runs when you insert the CD into the appropriate drive. The icon=*filename.ico* line specifies an icon to represent the AutoPlay-enabled CD in the Windows 95 user interface.

Opening a Startup Application

Most users need instant feedback when they insert an AutoPlay CD. You can accomplish this by using the open command of the Autorun.inf file to open a small startup executable (.exe) file that loads very quickly. The startup application should include the following features:

- Quick feedback to the user.
- Clear identification of the title to be played.
- Simple cancellation.

Loading in the Background

Generally, users will choose **OK** in the dialog box of the startup application. Take advantage of this by starting another thread that loads the setup application in the background before the user chooses **OK**. This significantly reduces the perceived load time for your application.

Conserving Hard Disk Space

Hard disk space is a limited resource. Here are a few tips that conserve hard disk usage:

- Run your application from the CD-ROM directly, without running any installation program.
- If your application needs to use the hard drive, install only the functional components necessary to run the application, and provide an uninstall function that removes the application from the hard disk. For more information about uninstalling an application, see the documentation included with the Microsoft Win32 Software Development Kit (SDK).
- If your application needs to use the drive as a data cache, provide the user with options in the startup application that will enable the cached data to be discarded when quitting the title or game.

Using the Registry

The registry is a feature of Windows 95 that supersedes the initialization (.ini) and application configuration files. For more information about registry manipulation application programming interfaces (APIs), see the documentation included with the Microsoft Win32 SDK.

If your title records and uses initialization information, you can use the registry to store and retrieve this information. Your startup application can use the information in the registry to determine whether the product needs to be installed:

- If your product is being used for the first time (no registry entries), present a dialog box that lists the setup options.
- If your product is installed (listed in the registry), present a dialog box with just the **OK** and **Cancel** buttons.

For debugging purposes, you can enable a machine to read the Autorun.inf file from a floppy drive by changing the system registry. Use the following procedure to do this:

1. Use the REGEDIT tool to search for the string "auto."

   The results of the search should be in the HKEY_CURRENT_USER key under the following subkey:

   `\Software\Microsoft\Windows\CurrentVersion\Policies \Explorer`
2. Change the data of the NoDriveTypeAutoRun value from 0000 95 00 00 00 to 0000 91 00 00 00.

   This change enables AutoPlay on any drive; however, you must select AutoPlay when working with a floppy disk to run the startup application. You can double-click the floppy disk icon, or select the floppy disk icon and access the shortcut menu to start AutoPlay.
3. After completing your tests of Autorun.inf, reset the value of NoDriveTypeAutoRun to 0000 95 00 00 00.

Important The capability of implementing AutoPlay on floppy disks is provided solely to help programmers debug their Autorun.inf files before burning the CD. AutoPlay is intended for public distribution on CD only.

There is an inherent danger of this technique spreading viruses through the floppy disks. Therefore, it is appropriate to suspect that any publicly distributed floppy disks that contain Autorun.inf files are contaminated.

Setting the NoDriveTypeAutoRun Value

The NoDriveTypeAutoRun value in the registry is a four-byte binary data value of the type REG_BINARY. The first byte of this value represents different kinds of drives that can be excluded from working with AutoPlay.

The initial setting for the first byte of this value is 0x95, which excludes the unrecognized type drive, DRIVE_UNKNOWN, DRIVE_REMOVEABLE, and DRIVE_FIXED media types from being used with AutoPlay. You can enable a floppy disk drive for AutoPlay by resetting bit 2 to zero, or by specifying the value 0x91 to maintain the rest of the initial settings. A table identifying the bits, bitmask constants, and a brief description of the drives follows:

Bit Number	Bitmask constant	Description
0 (low-order bit)	DRIVE_UNKNOWN	Drive type not identified.
1	DRIVE_NO_ROOT_DIR	Root directory does not exist.
2	DRIVE_REMOVEABLE	Disk can be removed from drive (a floppy disk).
3	DRIVE_FIXED	Disk cannot be removed from drive (a hard drive).
4	DRIVE_REMOTE	Network drive.
5	DRIVE_CDROM	CD-ROM drive.
6	DRIVE_RAMDISK	RAM disk.
7 (high-order bit)		Reserved for future use.

Suppressing AutoPlay

You can bypass the AutoPlay feature by holding down the SHIFT key when inserting a CD. This prevents Autorun.inf from being parsed and carried out.

AutoPlay for MS-DOS-Based Applications

AutoPlay can also be used to install, configure, and run MS-DOS-based applications in a Windows 95 MS-DOS session. Each MS-DOS-based application can be configured with its own unique icon, Config.sys file, and Autoexec.bat file.

Windows 95 creates correct configuration files for an MS-DOS-based application. The startup application can then cause the MS-DOS-based application to start within a window. It can even cause Windows 95 to effectively restart the application if necessary and immediately return to Windows when the application closes.

defaulticon

```
defaulticon=path\iconname.ico
```

Specifies an absolute path on the CD to locate the icon that represents the AutoPlay-enabled CD in the Windows 95 user interface.

path\iconname.ico

Absolute path and file name of the file containing the icon. You can also specify a .bmp, .exe, or .dll file. If a file contains more than one icon, specify the resource number (index) of the icon in the file to use.

If both icon and **defaulticon** commands are present in an Autorun.inf file, AutoPlay uses the icon specified in the **defaulticon** command.

When the drive does not contain an AutoPlay-enabled CD, the system uses a default icon in its place.

icon

```
icon=filename.ico
```

Specifies an icon to represent the AutoPlay-enabled CD-ROM in the Windows 95 user interface. The file name specified with this command must be located in the same directory as the file name specified by the open command.

filename.ico

Name of the file containing the icon. You can also specify a .bmp, .exe, or .dll file. If a file contains more than one icon, specify the resource number (index) of the icon in the file to use.

When the drive does not contain an AutoPlay-enabled CD, the system uses a default icon in its place.

The following example specifies the second icon in a file to represent a CD. The first icon's index is set to zero.

```
icon=filename.exe 1
```

open

```
open=dir\filename.exe
```

Specifies the path and file name of the application that AutoPlay runs when you insert the CD into a CD-ROM drive.

dir\filename.exe

Path and file name of any executable file to run or document to open when the CD is inserted. If no path is specified, Windows 95 looks for the file in the root of the CD. You can specify a relative path to locate the file in a subdirectory.

If the file is a document, Windows 95 starts the application associated with the specified document.

You can also include command-line parameters that AutoPlay passes to the program upon execution.

Use this command to open a startup application that provides instant feedback for the user. For more information about startup applications, see Opening a Startup Application.

shell

```
shell=verb
```

Changes the default entry of the shortcut menu to the specified custom command.

verb

Abbreviated form of a custom command. The custom command must be defined in the Autorun.inf file.

AutoPlay is the default menu item defined for any AutoPlay-enabled disk.

When the user double-clicks on the icon for your CD, the command associated with the verb is carried out.

shell\verb

```
shell\verb\command=filename.exe
shell\verb=Menu Item Name
```

Specifies a custom command listed in the shortcut menu for the icon. The first line identifies the executable file that performs the command. The second line specifies the custom entry of the shortcut menu.

verb

Abbreviated form of the command. This parameter associates a command with the executable file name and the menu item. It must not contain embedded spaces.

You will not see *verb* unless *Menu Item Name* is omitted from Autorun.inf.

filename.exe

File name of the application that performs the custom command.

Menu Item Name

Menu item text that can contain mixed case letters and spaces. You can also set an accelerator for the menu item by preceding one of the letters in the item with an ampersand.

When you point to an icon in the Windows 95 user interface and click the right mouse button, Windows 95 presents a shortcut menu for that icon. If an Autorun.inf file is present on a CD, Windows 95 automatically adds AutoPlay to the shortcut menu, and sets it as the default behavior. Double-clicking the icon will carry out whatever is specified in the open command.

To add the command ReadMe to the shortcut menu for your product, include the following in the Autorun.inf file:

```
shell\readit\command=notepad abc\readme.txt
shell\readit=Read &Me
```